Université des Sciences et Technologies de Lille 1

N° attribué par la bibliothèque :
3190

Année 2002

Thèse

En vue de l'obtention du grade de

Docteur de l'Université des Sciences et Technologies de Lille 1
Discipline : Informatique

Présentée et soutenue publiquement
par
Xavier LE PALLEC

le 04 décembre 2002

Des services d'adaptation de modèles pour la coopération de méta-systèmes : application aux groupware flexibles

Directeur de thèse :
Pr. A. Derycke

Jury

Pr. **J.-M. Geib**, D.R., Université de Lille I	Président du Jury
Pr. **A. Derycke**, Université de Lille I	Directeur
Pr. **M.-P. Gervais**, LIP6, Université Paris X	Rapporteur
Pr. **C. Godart**, ESSTIN, Université Henri Poincaré Nancy1	Rapporteur
Pr. **J. Malenfant**, Université de Bretagne sud	Examinateur
M. Belaunde, France Telecom R&D, Lannion	Examinateur

Table des matières

Introduction

L'informatique est tellement présente au sein des entreprises qu'il est difficile d'imaginer aujourd'hui une entreprise sans ordinateur. Au fil des années, l'informatique a permis aux entreprises d'améliorer leur productivité. Les suites bureautiques ont amélioré la production individuelle. Les gros systèmes ou mainframes ont fortifié la gestion des stocks, du fichier clients ... Les systèmes de workflow ou tout autres groupware (i.e. systèmes de gestion du travail de groupe) ont optimisé la coordination des tâches individuelles et le flux d'information entre celles-ci. Et enfin, la vidéo-conférence ou les éditeurs coopératifs de documents ont rendu possible le travail de groupe même quand les participants sont distants géographiquement.

La présence de l'informatique ne s'arrête pas aux frontières de l'entreprise, elle intervient aussi dans les relations que l'entreprise entretient avec d'autres organisations. La coopération avec d'autres entreprises est due au marché économique tendu. La sous-traitance réduit les coûts de production. De nouveaux partenariats permettent de proposer de nouveaux produits et/ou services et de répondre ainsi aux attentes changeantes des clients. Le coût des échanges nécessaires entre entreprises a malheureusement toujours été un frein pour le partenariat. L'apparition de l'EDI (Electronic Document Interchange) et plus tard de l'Internet a peu à peu diminué ce coût et a accentué la tendance aux partenariats. Néanmoins, si les solutions informatiques de coopération d'entreprises gèrent efficacement leurs échanges de données, elles ont plus de difficultés pour la synchronisation de leurs activités.

L'obligation qu'a une entreprise à être dynamique ne s'applique pas qu'aux relations extérieures. Elle doit aussi être capable d'optimiser sa chaîne de production, d'intégrer de nouvelles activités, de nouvelles compétences, ... Les politiques de management préconisent souvent un grand dynamisme. Les groupware qu'utilisent les entreprises pour gérer les processus de production sont souvent un frein à ce dynamisme : la modification des règles de coordination nécessite souvent l'intervention d'informaticiens. Récemment, des groupware plus flexibles sont apparus : avec ces derniers, une entreprise peut modifier elle-même les règles de coordination du système sans demander l'intervention d'un informaticien. Le dynamisme de l'entreprise n'est plus freiné.

Le caractère dynamique des nouveaux systèmes informatiques va, dans le contexte de la coopération, accroître la difficulté à les synchroniser : si les règles de coordination changent au sein des entreprises, comment va se maintenir la coordination entre entreprises en termes informatiques. À la suite d'une évolution des règles de coordination d'une entreprise, si la mise à jour de la passerelle informatique avec une entreprise partenaire nécessite l'intervention d'informaticiens, le partenariat sera effectif plus tardivement. La flexibilité qu'a apporté les nouveaux systèmes informatiques perd alors de son intérêt. La synchronisation des systèmes doit pouvoir être effectuées par les entreprises elles-mêmes. Aucune solution actuelle d'interopérabilité n'offre une telle possibilité. C'est ce que nous proposons dans ce manuscrit.

Cette thèse, effectuée au sein de l'équipe NOCE (Nouvelles Organisations pour la Coopération et l'Éducation) du laboratoire Trigone, et sous la direction du Professeur des Universités, Monsieur Alain Derycke, propose une solution à la coopération de systèmes flexibles, et plus particulièrement ceux destinés au travail de groupe. Le laboratoire Trigone effectue des recherches sur le TCAO (Travail Coopératif Assisté par Ordinateur). ODESCA et DARE sont des systèmes développés au sein de ce laboratoire dont le but est de supporter des activités coopératives. La nécessité de flexibilité et de coopération inter-organisationnelle a déjà été mise en avant par les recherches sur les sciences sociales à la base de ces systèmes. Les préoccupations des entreprises l'ont confirmé.

En parcourant ce manuscrit, le lecteur pourra découvrir les différentes facettes de notre proposition. Nous définissons d'abord les contraintes que doivent respecter un support informatique à

la coopération de systèmes flexibles. Notre étude portent ensuite sur les méta-groupware. Ce sont des systèmes dédiés au travail de groupe où les utilisateurs peuvent modifier les règles de coordination (et de coopération). Nous proposons un outil pour la création de services d'adaptation (CAST) qui permettent à un méta-groupware d'adapter et d'intégrer les règles de coordination de meta-groupware externes. Les services d'adaptations permettent ainsi aux utilisateurs de synchroniser la coordination de leur système avec celle d'un autre système. Nous définissons d'abord un cadre conceptuel de l'outil. Nous l'implémentons ensuite sur le support MOF (Meta-Object Facility). Cette réalisation a nécessité le développement d'un outil de prototypage dédié au MOF : RAM3 (Rapid Manipulation of Mof Metadata).

Les chapitres de cette thèse exposent en détail l'ensemble de notre démarche et de nos apports. Ils sont agencés comme suit.

L'informatique au sein des entreprises a connu une évolution importante, que ce soit au niveau du travail individuel ou de la gestion du travail de groupe et de l'organisation. Actuellement, nous pouvons discerner deux évolutions vis-à-vis des systèmes : le besoin de coopérer avec des systèmes externes et la nécessité de devenir de plus en plus flexibles. L'objet du **chapitre 1** est de mettre en exergue un nouveau type de problème : la coopération de systèmes flexibles. Ce problème décrit, nous élaborons les contraintes à respecter pour toute solution à ce problème. Nous évoquons aussi l'intégration de ces travaux dans l'équipe NOCE.

Le chapitre deux a pour but d'étudier quelles solutions peuvent exister pour notre problématique. Il présente à cet effet les différentes solutions d'interopérabilité actuelles. Des critères d'évaluation sont présentés et mises en correspondance avec nos contraintes de départ. Elles nous permettent de juger de la pertinence des solutions, comme l'utilisation de standard ou de médiateurs, vis-à-vis de notre problématique. L'apport principal du **chapitre 2** est de démontrer que les solutions d'interopérabilité actuelles sont "conceptuellement" inadaptées à la coopération de systèmes flexibles.

La coopération entre deux organisations peut s'apparenter à la coopération au sein d'une organisation plus grande, union des deux précédentes. Lors d'une coopération entre deux organisations, les règles de coordination des tâches et de coopération des individus d'une des deux organisations doivent donc être modifiées de manière à être associées aux règles analogues de la seconde organisation (et vice-versa). La gestion du précédent type de règles est généralement l'objectif d'un groupware. Ceci entraîne que la coopération entre deux organisations se réalise plus naturellement par ce type de système (si les organisations en sont pourvues). Pour cette raison, nous spécialisons notre étude vers les groupware flexibles et analysons les moyens éventuels pour lier ce type de groupware afin de permettre une coopération inter-organisationnelle. L'objectif du **chapitre 3** est donc de définir précisément l'enjeu de cette thèse. Après une rapide présentation des groupware, nous insistons sur l'approche la plus efficace pour construire un groupware flexible – l'approche par méta-modèles. Ce chapitre se conclut par la démonstration de l'inadaptation "technique" des solutions d'interopérabilité à une telle approche. La présentation des groupware consiste principalement à décrire les systèmes de workflow (majoritaires dans ce domaine). Néanmoins nous insistons sur la spécialisation actuelle de tels systèmes et l'existence d'autres approches pour la gestion du travail de groupe (notamment illustrées dans le scénario exemple). *Ceci nous oblige à adopter une approche suffisamment générique pour qu'elle puisse fonctionner avec des systèmes de workflow mais aussi avec des systèmes comme DARE qui n'ont pas du tout les mêmes bases conceptuelles. C'est pour cette raison que notre solution portera sur le dénominateur commun à ces groupware flexibles : l'approche par méta-modèles. Elle n'intégrera pas de concepts spécifiques à certains type de systèmes (de peur d'avoir ensuite une portée limitée).* La description ("sommaire") des systèmes de workflow est utile pour donner matière à réflexion et illustration.

Notre solution consiste à placer des services d'adaptation pour effectuer :

- des traductions entre modèles de manière à ce que les utilisateurs voient et puissent manipuler des éléments (de modèles) d'autres systèmes, et ainsi les associer à des éléments de leur système,

- des traductions entre instances/réalisations des modèles afin de *permettre* aux liens entre modèles de synchroniser les instances/réalisations.

Le **chapitre 4** présente notre proposition – CAST (Creation of Adaptation Service Tool) – et expose son cadre conceptuel, c'est-à-dire sans se soucier du support informatique à la description des modèles et de leurs réalisations/instances. CAST a pour objectif de permettre une construction rapide de services d'adaptation (qui viennent ensuite lier deux systèmes). Dans ce chapitre, y est défini le fonctionnement de CAST et son formalisme graphique pour définir les liaisons sémantiques entre deux méta-groupware (i.e. des groupware adoptant l'approche par méta-modèles). Le modèle conceptuel générique d'un service d'adaptation généré à partir des liaisons sémantiques fait guise de conclusion.

Notre approche ne peut-être validée qu'avec une réalisation pratique. Pour cela, nous choisissons un support informatique pour l'écriture des modèles et de leurs instances : le MOF (Meta-Object Facility). C'est parce que le MOF est pratiquement le seul standard de représentation de modèles (de tout types) qu'il constitue notre choix. Dans le **chapitre 5**, nous décrivons précisément ce récent standard destiné à la description de modèles. Ceci nous permet ensuite de détailler l'implémentation du modèle conceptuel de CAST sur le MOF : M-CAST. Par la même occasion, nous définissons des extensions au MOF qui n'est pas suffisamment évolué pour inclure des aspects propres aux méta-systèmes (notamment les instances de modèles).

Mettre en place un service d'adaptation demande des précautions : celui-ci doit être testé suffisamment pour éviter toute erreur sémantique qui entraînerait des incohérences et donc des pertes financières. C'est le principal problème de la réalisation de M-CAST, objet du **chapitre 6**. Il nous fallait un outil pour prototyper les services d'adaptation qui au final ne sont que des "objets" MOF dont le comportement est d'adapter. Les outils MOF actuels se limitent à créer des fabriques de modèles pour un formalisme donné. Ils ne disposent pas de facilité de prototypage. Ils ne peuvent pas servir à CAST pour prototyper nos précédents objets MOF d'adaptation. Pour cela, nous avons décidé de développer un outil MOF – RAM3 (RApid Manipulation of Mof Metadata). Il propose comme tout outil MOF de créer des fabriques de modèles MOF mais apporte en plus un environnement pour prototyper ces fabriques. Il a comme seconde particularité d'implémenter les précédentes extensions "conceptuelles" que nous avons apportées au MOF ce qui en fait un véritable environnement de développement de méta-systèmes compatibles MOF (les outils MOF actuels ne gérant pas les instances de modèles). C'est ce qui a motivé en partie le développement de RAM3 : favoriser l'approche par méta-modèle et le MOF (deux choix à la base de nos travaux). Enfin, sa modularité, l'intégration aisée de code d'implémentation et son assistance à la création d'interface graphique en font un excellent support au développement de M-CAST.

Le **chapitre 7** résume notre démarche et expose les apports de CAST et RAM3 dans différents domaines. Les perspectives de travaux de recherches futurs clôturent cet ouvrage.

Il est très important de noter que le travail présenté ici a nécessité de toucher à différents domaines de la science informatique et au-delà : les organisations virtuelles, l'interopérabilité de systèmes informatiques, la méta-modélisation, les systèmes réflexifs, les systèmes de workflow, les systèmes orientés coopération, les applications réparties et les sciences sociales. Bien évidemment, il a fallu se concentrer sur une des facettes (plutôt les problèmes d'une approche générique d'interopérabilité par les méta-modèles) et il est donc évident que ce travail ne prétend pas à une vue exhaustive et très experte des autres domaines étudiés. *Notre solution a d'ailleurs un focus plus large : l'interopérabilité de systèmes adoptant l'approche par méta-modèles. Pour notre problématique, cette solution est appliquée aux groupware (même si l'approche par méta-modèles est essentiellement utilisée par les groupware).* Une solution spécifique, par exemple, au domaine du workflow

nécessiterait un travail, à partir de notre proposition, avec des équipes de recherche spécialisées. Il en est de même pour les aspects de dynamique inter-organisationnelle et de coopétition [GOD 99].

Note au lecteur : les parties ou sections dont le titre est accompagné d'un astérisque (*) peuvent être qualifiées de *techniques*. Elles peuvent donc être survolées.

Chapitre 1

Un besoin grandissant d'interopérabilité et Une évolution des systèmes

Les systèmes qui fournissent un support informatique aux systèmes d'information doivent de plus en plus coopérer entre eux. Si cette nécessité a toujours existé, elle est aujourd'hui plus importante. Les stratégies économiques, comme le partenariat, accentuent en effet cette exigence. Parallèlement, le dynamisme du marché économique exige de ces systèmes une plus grande capacité d'adaptation. La flexibilité devient aujourd'hui un nouvel argument de vente pour les systèmes d'entreprise : modification de l'ordonnancement des tâches, des droits d'accès aux documents, de la régulation, intégration de nouvelles fonctionnalités... La flexibilité entraîne des propriétés qui viennent complexifier la coopération entre de tels systèmes. Le caractère récent de l'apparition de systèmes industriels 'très' flexibles implique que cette complexité n'a pas encore été réellement traitée.

L'objectif de ce chapitre est de désigner la problématique générale de cette thèse, c'est-à-dire le point de départ de nos travaux. Dans un premier temps, nous décrivons l'évolution de la demande vis-à-vis des systèmes informatiques : une plus grande capacité à coopérer et une plus grande flexibilité. D'un point de vue conceptuelle, nous évoquons ensuite les caractéristiques propres à un système flexible. Cette évocation nous permet finalement de démontrer, contraintes à l'appui, la complexification de la coopération entre systèmes informatiques, quand ces derniers sont ou seront flexibles.

Notre conclusion porte sur l'utilité de ces travaux dans le domaine du Travail Coopératif Assisté par Ordinateur – TCAO – (au centre des travaux de notre laboratoire).

1 Évolution de l'informatique dans les systèmes d'information

Les systèmes informatiques sont de plus en plus présents au sein des organisations. Leur apport permet aux entreprises d'améliorer leur capacité de communication, leur réactivité, leur productivité... Ces améliorations apportent de nouvelles demandes des entreprises vis-à-vis des concepteurs de systèmes informatiques, notamment un meilleur support à l'activité (individuelle ou collective) et des possibilités accrues de coopération inter-organisationnelle.

Chaque entreprise est un système de travail (*work system*) [ALT 00] où, à travers l'activité d'êtres humains, des produits et/ou des services vont être fournis à des clients internes ou externes. Pour cela, les participants utilisent des informations, des machines et d'autres ressources. Une partie de ce système de travail est le système d'information (SI). Il s'occupe de tout traitement de l'information (capture, transmission...). L'informatique est généralement sollicitée pour automatiser le SI.

Les premiers systèmes informatiques dédiés aux SI avaient pour objectif de régir les aspects administratifs des entreprises. La tâche des utilisateurs de tels systèmes était de fournir les informations nécessaires au système pour, par exemple, gérer les stocks, les salaires, ...

Techniquement, ils s'agissaient de mainframes ou de mini-ordinateurs associés à un système de gestion de base de données (SGBD) et un ou plusieurs terminaux (pour le ou les utilisateurs).

Avec la démocratisation de l'informatique personnelle et l'arrivée d'ordinateurs personnels bon marché, l'informatique a commencé à être utilisée pour améliorer la production individuelle : les traitements de texte ont accéléré la rédaction de documents, les logiciels de statistiques ont facilité les études de marché, etc. Parallèlement, les entreprises ont souhaité que les systèmes informatiques améliorent le travail de groupe. Des systèmes ont été développés pour coordonner l'ensemble des tâches individuelles ou d'une façon plus générale de gérer les procédés métiers[1]. Dans le domaine de la *coordination*, se trouvent principalement les systèmes de workflow (WfMS : *Workflow Management System*). Conjointement, des applications ont été conçues pour permettre le travail *collaboratif* : des logiciels de co-édition, de vidéoconférence ou des tableaux blancs partagés (applications que l'on appelle *collecticiel*). L'ensemble des applications et des systèmes qui s'occupent du travail de groupe est aujourd'hui associé au terme *groupware*[2].

Dans un système d'information, l'informatique devient présente à tous les niveaux. J.Grudin [GRU 94] illustre cet état de fait avec une représentation en trois anneaux (cf. fig1) :

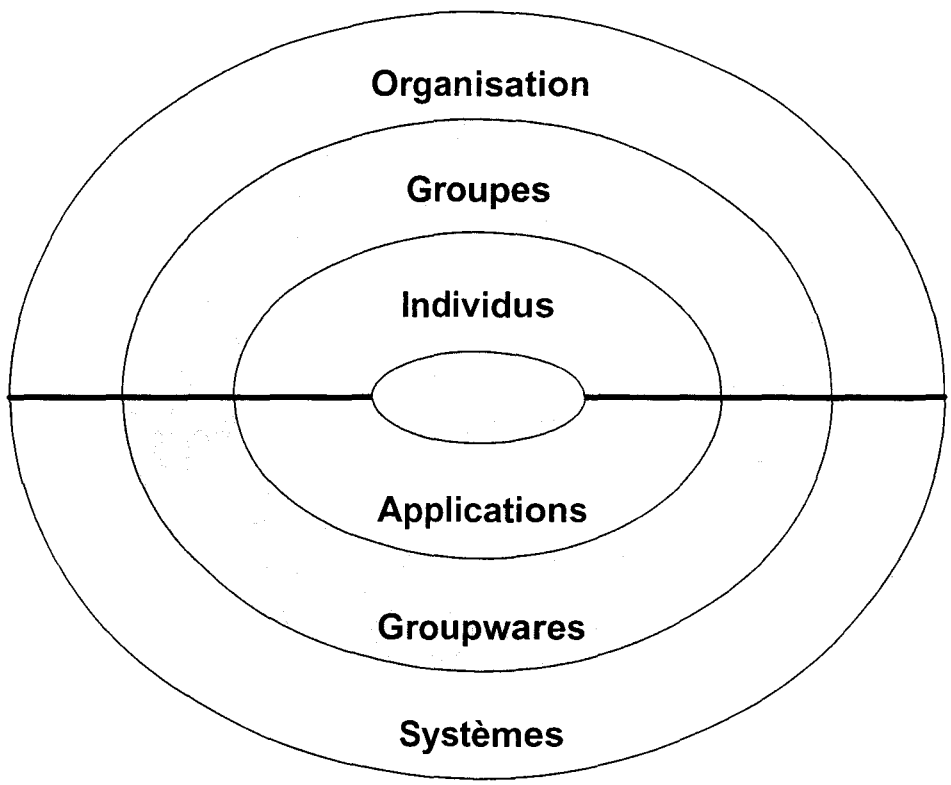


figure 1. La classification des activités de l'informatique

- Au niveau organisationnel : les systèmes qui gèrent les caractéristiques de l'organisation. Pour une entreprise, ces caractéristiques sont par exemple le fichier client, les stocks, mais aussi, pour les systèmes plus évolués, sa politique et ses objectifs commerciaux (*business goals*) [DEM 97].

[1] "*Une collection d'activités consommant des entrées (matériels, finances, données, ...) et délivrant un ou plusieurs résultats à orientation économique ou à forte valeur ajoutée pour l'entreprise*" [SAI 01]

[2] Bien que ce terme fût au départ associé aux systèmes plus informels et orientés communication en opposition aux systèmes de workflow. Il s'agissait justement de systèmes visant à libérer les utilisateurs de WfMS de la rigidité de ces derniers [HOE 01].

- Au niveau collectif : les groupware qui gèrent les flux au sein de l'organisation, le travail inter-individuel. Ces groupware peuvent suivre des modèles ou règles de coordination, communication et coopération.
- Au niveau individuel : il met à disposition des utilisateurs des outils qui facilitent le travail.

Le support "informatique" est aujourd'hui une pièce maîtresse des systèmes d'information. Il accompagne l'entreprise dans chacune de ses activités. De ce fait, les pressions du marché économique qui s'exercent sur les entreprises ont des répercussions sur les systèmes informatiques. Le besoin de coopération entre entreprises est une de ces pressions. La répercussion directe sur les systèmes informatiques est la nécessaire présence de mécanismes d'interopérabilité.

2 Coopération d'entreprises et Interopérabilité

Le marché économique actuel est le siège d'une compétition féroce. L'arrivée de la Chine, des pays de l'ex-Union Soviétique et des pays du tiers-monde en est, en partie, à l'origine. La stagnation du marché dans les pays développés (les biens ne sont plus aujourd'hui à acquérir mais à renouveler) et l'amélioration technologique des réseaux de communication sont sans doute les autres raisons majeures de cette compétition. Pour faire face, deux stratégies ont été adoptées : la fusion et le partenariat. La fusion d'entreprises permet d'avoir un poids important sur le marché et le partenariat permet aux entreprises de construire des entreprises virtuelles capables de s'adapter aux fluctuations du marché. Nous nous intéressons ici aux entreprises virtuelles et aux relations inter-organisationnelles qu'elles mettent en jeu. Le concept d'entreprises ou d'organisations virtuelles (VO : *virtual organization*) est apparu à une période où les organisations ont ré-examiné leur stratégie et leur structure dans le but de survivre dans un nouvel environnement international (compétitif) [KLE 96]. Une VO est définie comme "*a temporary consortium of independent member companies coming together to quickly exploit fast-changing world-wide product manufacturing opportunities*" [HAR 97]. En d'autres termes la pratique du partenariat permet à des entreprises de profiter de nouvelles opportunités commerciales (*time-to-market*). Pour fournir un produit ou un service, une entreprise a besoin d'un certain nombre de compétences et de ressources. Par exemple, pour construire une maison, elle doit employer des maçons, des plombiers, des couvreurs, ... et disposer de bétonneuses, de camions, etc. Un particulier ne peut donc pas demander à une simple société de plomberie de construire une maison. Par contre, cette société, pour saisir l'opportunité commerciale peut s'associer à des sociétés de maçonnerie, de peinture, ... et se proposer comme constructeur [GEO 99]. De manière générale, le partenariat aide une entreprise à s'insérer dans de nouveaux marchés. L'enrichissement accumulé lors d'une coopération apporte aussi de nouvelles idées de collaboration [KLE 96]. Il est à noter que la coopération n'implique pas une absence de compétition : deux sociétés peuvent coopérer pour un projet et être en compétition pour un autre (d'où la notion de *coopétition* [CAN 02]). La coopération permet simplement d'élargir la clientèle et de s'adapter aux fluctuations du marché[3].

La coopération a néanmoins un coût : l'échange d'information (dupliquer les plans), la coordination (employer un coordinateur), ... En fait, le partenariat devient intéressant si les bénéfices perçus sont plus importants que les coûts de coopération [SUB 00][4]. L'informatique peut être perçue comme un moyen de diminuer ces coûts. Néanmoins informatisation n'implique pas nécessairement une baisse des coûts dans ce type de transaction. Longtemps les échanges entre systèmes informatiques ont été des échanges papiers (ressaisie) ou sur support magnétique. Même si le dernier support est moins coûteux car plus compact et plus rapide que le premier (il évite une nouvelle saisie), son transport, par voie postale ou par courrier, reste néanmoins lent. Dans les échanges papiers (les plus nombreux), de nombreux documents font double emploi (ex : un bon de commande et un bon de

[3] Un autre avantage des VO est de permettre aux micro-entreprises de devenir plus importante (PME) [SER 96].

[4] Il est d'ailleurs difficile d'évaluer les réels bénéfices d'une coopération : à partir de quand comptabilise-t-on une dépense dans le contexte d'une coopération ?

livraison contiennent normalement exactement les mêmes informations)[5]. Enfin, lors des ressaisies, les erreurs humaines sont fréquentes et lourdes en conséquences. Pourtant l'informatique dispose d'un fort potentiel comme moyen de diminuer les coûts de coopération. Pour cette raison, les industriels ont rapidement décidé d'utiliser ce potentiel : échanger les documents sous format électronique à travers des lignes téléphoniques ou des lignes spécifiques. C'est à la fin des années 50, que naît l'EDI (Electronic Data Interchange) sous l'impulsion du secteur des transports (car les échanges d'information, comme les réservations, y sont nombreux). L'EDI est défini comme *le remplacement des documents papiers utilisés dans l'administration, le commerce et les transports par des messages numériques d'un ordinateur à un autre sans aucune intervention humaine* [SIT 89]. Un des objectifs de l'EDI est de normaliser le format des échanges électroniques selon les secteurs. Même s'il permet de gagner du temps (en transfert) et de l'argent (gain de temps et coût papier), l'EDI n'est pas un succès. Aujourd'hui seul 10% des compagnies des États-Unis (et 5% dans le monde) utilisent ce système [OMA 00]. Le principal coupable de cet échec sont les supports de transport (notamment les RVA : Réseau à Valeur Ajouté) dont la mise en place et l'abonnement sont coûteux. C'est l'apparition de l'Internet qui apporte une alternative peu onéreuse à EDI[6] et instaure réellement le règne des échanges électroniques. L'utilisation est bien moins onéreuse que les supports EDI et les contraintes de sécurité et de légalité ne sont pas obligatoires (contrairement aux échanges EDI). Les petites sociétés et les PME adoptent très vite ce support, d'où le succès grandissant du commerce électronique B2B (Business to Business)[7].

L'apparition d'un support d'échange de documents rapide et "peu" coûteux favorise le partenariat d'entreprises. Néanmoins, la possibilité qu'ont les systèmes informatiques à échanger directement des informations n'automatise pas complètement la coopération entre entreprises. Pour un même document reçu, un système informatique peut, selon le contenu ou le contexte, réagir de différentes manières. L'automatisation des échanges nécessite donc d'intégrer aux systèmes les règles de la coopération. Sur l'exemple d'entreprises employant des systèmes de workflow (WfMS), l'intégration des règles correspond à la synchronisation des (modèles de) procédés métiers que gèrent ces systèmes.

Si Internet a rendu accessible à tous les échanges électroniques, il a aussi accentué le problème de la diversité des formats de documents déjà apparu avec l'EDI (mais moins important à cause de la présence forte d'organisations de normalisation). L'échange d'information entre systèmes informatique implique donc souvent des processus de conversions des documents. Ces conversions ont aussi un coût (ex : la création d'un programme spécifique) qu'il est critique de diminuer [KAT 98]. Cette différence de format apparaît aussi au niveau de la coordination des procédés métiers qui sont définis dans des formalismes différents (d'où l'initiative du WfMC[8]). Les règles de coopération inter-systèmes, bâties sur les modèles de coordination des systèmes coopérants, en sont donc complexifiées.

Pour cette raison un grand nombre de travaux sont effectués sur le problème de la coopération entre systèmes informatiques (cf chapitre 2) et des mécanismes inter-organisationnelles à mettre en place. Le terme "'interopérabilité" est souvent employé mais il désigne un domaine plus vaste (fusion de systèmes, intégration, ...). Les recherches sur l'interopérabilité ont débuté il y a plus d'une vingtaine d'années [PAE 98]. L'utilisation de normes, les ontologies, les intergiciels ou les solutions à base de médiateurs sont différentes approches grâce auxquelles des systèmes sont capables d'interopérer. Ces approches sont bien sûr ré-utiliser pour la problématique de coopération inter-organisationnelle. Les

[5] Par exemple, en 1990, le coût "papier" sur des échanges internationaux revient à 7 %.

[6] Bien qu'actuellement un certain nombre de fournisseurs de produits EDI fournissent une version Internet. Malheureusement cette migration est ralentie à cause des grosses entreprises (les pionners de l'EDI) qui souhaitent continuer, de par les lourds investissements de départ, à utiliser les réseaux habituelles : RVA, lignes privés ou lignes publiques (ex : Transpac).

[7] En 1999, le B2B représente 80 milliards de dollars sur les 111 correspondant au commerce électronique (source http://www.it-director.com).

[8] Le Workflow Management Coalition a pour but de proposer un formalisme standard pour la définition des processus workflow.

solutions existantes sont adaptées aux structures des systèmes qu'elles font interopérer. Ces structures évoluent actuellement pour rendre les systèmes plus flexibles.

La flexibilité est une propriété qui a récemment été mise en avant par des différents travaux notamment dans le domaine du workflow et du TCAO. La section suivante présente les raisons qui amènent les systèmes à être plus flexibles. Nous verrons dans la section 4 que la flexibilité apporte de nouvelles contraintes dans la coopération inter-organisationnelle et oblige à concevoir de nouveaux mécanismes respectant ces contraintes.

3 La Flexibilité : une nouvelle exigence pour les systèmes informatiques

Gérer les informations et la coordination des tâches individuelles sont deux des principaux objectifs des systèmes informatiques d'entreprises conçus jusqu'au milieu des années 90 [SAI 01]. Les différentes observations, commentaires et souhaits d'améliorations apparus pendant l'utilisation de ces systèmes en entreprise, dresse un cahier des charges pour les systèmes à venir [SHE 97]. Une plus grande flexibilité est une caractéristique majeure de ces futurs systèmes.

3.1 Une nécessité d'adaptation et d'optimisation

Nous avons déjà évoqué l'importante compétition qui règne dans le marché économique actuel. La structure modulaire, appelée aussi *réseaux de valeurs*, construite grâce au partenariat (évoqué en 2), apporte une grande capacité d'adaptation aux entreprises. Celles-ci s'occupent alors d'un stade précis de production [KOC 00][KLE 96]. Les associations n'ont qu'à se défaire lorsqu'un marché disparaît ou n'est plus intéressant. A l'inverse, une société qui gère tous les stades de production doit effectuer des modifications lourdes dans sa structure quand elle le peut ou disparaître sinon[9]. La pratique du partenariat, évoquée précédemment, ne met pas à l'abri de la compétition. Une société doit rester un partenaire attrayant : il doit être bon marché (et donc supprimer les coûts de production inutiles), être au courant des dernières technologies, s'adapter à toutes nouvelles tendances... L'optimisation et l'adaptation demeurent des lignes de conduite à suivre.

3.2 Agilité, flexibilité, ...

Pour optimiser ses traitements (efficacité) et s'adapter aux fluctuations du marché (opportunisme), une entreprise doit être *agile*. Dove [DOV 97] définit l'agilité d'une entreprise comme la capacité à gérer les changements de manière à saisir les opportunités aussi bien qu'à innover. Dove en déduit un certain nombre de caractéristiques : variation des ressources utilisées, tolérance aux fautes, capacité de réorganisation, d'amélioration, d'intégration, ... Les systèmes informatiques ont prouvé leur capacité à gérer les informations et les procédés métiers d'une entreprise. Il leur est aujourd'hui demandé d'être eux-mêmes agiles afin de ne pas réduire l'agilité de l'entreprise.

Différents termes peuvent évoquer l'agilité pour les systèmes informatiques. La littérature est prolifique dans ce domaine : adaptabilité, flexibilité, malléabilité, extensibilité, expansivité... Comme souvent en informatique, il n'y a pas de définition standard pour chacun de ces termes. Dans certains cas, la flexibilité inclut l'adaptabilité [SAI 01], dans d'autres, l'adaptabilité inclut la flexibilité [FYA 96]. Pour Grøbæk [GRØ 94], beaucoup de ces termes signifient la même : permettre aux utilisateurs de modifier eux-mêmes leur système. En d'autres termes, les modifications du système doivent éviter au maximum l'intervention des concepteurs. Néanmoins, nous souhaitons apporter une nuance pour chacun des termes que nous utiliserons dans la suite de ce mémoire.

[9] D'ailleurs, pour réagir, des grandes compagnies (e.g. Sony, ATT, GMC) se sont subdivisées [KOC 00] en petites sociétés. Ces dernières, même si elles restent attachées à une autorité supérieure, essayent d'agir comme des sociétés indépendantes (ex : partenariat avec des sociétés extérieures).

- La flexibilité : Elle insiste sur la modification du système (qualité). Plus celui-ci est flexible, moins les opérations de modifications consomment de temps.
- L'extensibilité : Elle privilégie l'ajout de composants aux systèmes (quantité) ou la connexion avec d'autres systèmes.
- La malléabilité : elle focalise sur la facilité d'accès pour les utilisateurs 'finaux' aux précédents mécanismes (l'un et/ou l'autre).
- L'adaptabilité : elle fait parfois apparaître la capacité de réaction aux événements imprévus (tolérance aux pannes/fautes).

Ces nuances n'ont pas pour but de faire figure de définitions standard mais simplement d'établir des définitions pour la suite de ce mémoire. Les systèmes informatiques agiles ne supportent pas qu'une seule des caractéristiques précédentes mais privilégie généralement l'une d'entre elles (pour se démarquer commercialement). Nos travaux se sont focalisés sur la flexibilité et les structures la mettant en œuvre. Nous verrons dans la partie 4 qu'elle remet en cause la façon d'implémenter la coopération avec d'autres systèmes.

3.3 La flexibilité dans les systèmes informatiques d'entreprise

Le système informatique d'une entreprise a pour but de gérer, d'automatiser et d'améliorer tout ou partie de son système d'information. La volonté de disposer de systèmes informatiques flexibles provient de la dynamique nécessaire des entreprises et donc de leur système d'information. Nous présentons, de manière générale, les points de flexibilité possibles ou souhaités d'une entreprise et leurs répercussions sur le système informatique. Nous nous inspirons pour cela des travaux réalisés par les fondateurs de la communauté CoopIS [DEM 97]. Ils proposent un cadre d'étude pour analyser l'évolution des nouveaux systèmes informatiques (pour les systèmes d'informations). Ils considèrent que les futurs systèmes seront généralement composés d'un agrégat d'applications ou de systèmes coopérant[10]. Leur cadre vise à étudier les scénarios d'évolution d'un tel système par rapport aux demandes de l'organisation. Nous utilisons ce point de vue pour faire une présentation générale de la flexibilité dans le système informatique.

On peut décomposer un système d'information en trois facettes :

- **Collaboration de groupe.** Cette facette décrit la façon dont les gens travaillent ensemble, coordonnent leurs activités, gèrent les contingences et peuvent changer leurs pratiques à travers des discussions ou l'apprentissage.
- **Organisation.** Elle définit les préoccupations globales, c'est-à-dire les objectifs de l'organisation, les buts commerciaux, les politiques, les régulations... et donc l'insertion dans de nouveaux marchés, la production de nouveaux produits...
- **Système.** Il s'agit du support informatique. S'y trouvent le ou les systèmes utilisés (SGBD, WfMS, progiciels), les applications utilisées (dont une partie sont les outils proposés aux employés), les types d'informations manipulés, les langages de programmation employés... Seul le côté technique doit être pris en compte dans cette facette. Ce qui implique son absence dans les deux autres facettes. Les mots "système informatique d'une entreprise" regroupent l'ensemble des précédents éléments qu'une entreprise contient.

[10] La gestion de cette agrégation est déjà l'objet des solutions d'EAI (Enterprise Application Integration). L'agrégation est souvent causée par les fusions qu'à effectué l'entreprise, l'augmentation de son parc informatique à la suite de l'apparition d'une nouvelle activité, ...

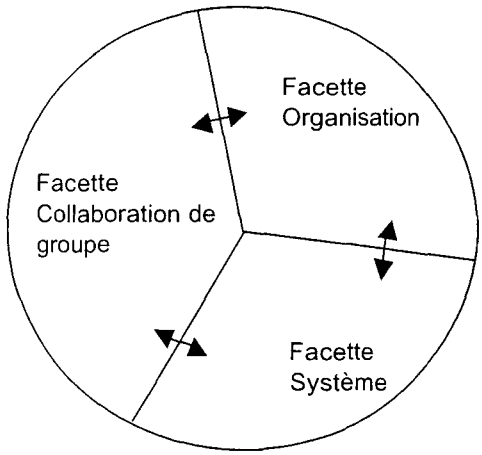


figure 2. Les trois facettes d'un système d'information

Ces facettes sont fortement liées (cf. figure 2). Toute modification dans l'une d'elles a des répercussions sur les deux autres. Le système peut donc être modifié pour des raisons internes mais aussi en conséquence d'une modification de l'aspect "organisationnelle" ou "collaboratif". Sa flexibilité est en rapport avec sa capacité à pouvoir supporter le plus facilement ces modifications.

3.3.1 Supporter les modifications de l'organisation.

En cas d'insertion dans un nouveau marché, l'entreprise propose de nouveaux types de produits ou de services. Le système informatique peut-être logiquement amené à supporter de nouveaux types d'information (ex : pour décrire le nouveau type de produit) ou de document (ex : supporter un autre format EDI). L'intégration d'applications spécifiques à ces nouveaux produits ou services est une éventualité fort probable. Les types d'information peuvent aussi évoluer sous l'impulsion d'une volonté de l'organisation à modifier ses services par exemple pour des raisons de rendement.

La création, modification ou suppression de types d'information et l'intégration d'application doivent pouvoir s'effectuer rapidement, dynamiquement et si possible par les "spécialistes" (ex : responsables du SI, du management,...) de l'entreprises.

De nouveaux services ou une évolution de ceux déjà présents entraîne très souvent une modification dans la coordination des activités ou/et la collaboration des employés (facette collaboration). Une telle modification peut aussi provenir du changement des unités structurelles de l'organisation ou des relations entre celles-ci. Ces répercussions sur la facette collaboration a elle-même des répercussions sur le système informatique.

3.3.2 Supporter les modifications de la collaboration de groupe

À la suite d'un changement dans la politique de l'organisation, la coordination des activités de l'entreprise et la collaboration des employés peuvent être amenées à évoluer. Cette évolution peut aussi résulter d'une plus grande expérience des employés qui améliorent leur façon d'effectuer leurs activités, répartissent différemment les tâches ou définissent des "raccourcis". La gestion des groupes étant de plus en plus supportée par le système informatique, les spécialistes (employés responsables du management) doivent pouvoir lui indiquer ces changements de manière à ce qu'il modifie les couloirs de transfert de documents, qu'il distribue les tâches selon les nouvelles règles, ... Des moteurs génériques de coordination qui peuvent exécuter différents modèles, règles ou spécifications sont des réponses techniques possibles pour supporter de tels changements.

Dans la coordination des activités, des erreurs peuvent survenir (le blocage d'un traitement, d'une machine), des situations indésirables (une date d'échéance non respectée) ou non prévues. La gestion de ces situations exceptionnelles est stratégique pour une entreprise. Si le système

informatique a le rôle de coordinateur, il a comme obligation de gérer ces exceptions. La flexibilité d'une telle fonctionnalité est, par exemple, de proposer aux utilisateurs d'indiquer les erreurs susceptibles d'intervenir et les actions à effectuer dans ce cas. Ces actions peuvent être des demandes d'intervention humaine [BLY 97].

La répartition 'intelligente' des tâches suivant la charge de travail des employés ou des sites informatiques est aussi une qualité souhaitée par les entreprises vis-à-vis de leur système informatique. La présence de paramètres peut apporter une certaine flexibilité aux mécanismes de régulation.

Enfin, il faut prendre aussi en compte les collaborations synchrones où le système informatique peut être un des supports de médiation (ex : vidéo-conférence, éditeur de texte coopératif). La création de nouvelles activités coopératives (synchrones) ou la modification de celles existantes peut impliquer l'intégration de nouveaux outils coopératifs (ou médias), de nouvelles règles de partage des outils coopératifs... Ce sont d'autres points de flexibilité que peut proposer le système informatique.

3.3.3 Supporter les modifications propres au système

Les changements dont l'origine est liée au système peuvent être d'ordre technologique. L'adoption d'une nouvelle interface de communication, comme les web services, l'utilisation d'un nouveau système d'exploitation, l'acquisition de nouvelles plate-formes matérielles en sont quelques exemples. Il est très difficile de proposer des points de flexibilité dans ce domaine. L'intervention d'informaticiens est quasi obligatoire.

Les modifications peuvent aussi être d'ordre plus applicatif (destiné au travail individuel) : le changement d'une interface graphique, l'ajout de composants (traitements, fonctions), etc. Il est plus facile dans ce cas de fournir des points de flexibilité. L'ajout du module de gestion de bibliographie End-Note à Microsoft Word, est réalisable par un utilisateur. Le paramétrage de l'interface de Word l'est aussi. Les possibilités de changement accessibles aux utilisateurs sont très variables.

Enfin il y a la capacité d'intégrer d'autres applications ou d'autres systèmes. Un grand nombre de travaux, comme les plate-formes "composants", les intergiciels (Corba, WSDL/SOAP...) ou les systèmes à base de connaissances, ont été et sont effectués pour faciliter de telles intégrations. En général, ces technologies sont principalement (pour l'instant) une aide aux développeurs. Une grande flexibilité n'est pas encore d'actualité dans ce domaine.

3.3.4 La flexibilité du système informatique

La liste des points possibles de flexibilité d'un système informatique est la suivante :

- Évolution (ajout, modification, suppression) de la structure des informations ou documents manipulés.
- Évolution de la coordination des activités.
- Évolution des fonctionnalités des applications.
- Évolution des règles de partage d'outil coopératif (et d'information).
- Gestion des situations exceptionnelles (évolution de la politique de gestion).
- Gestion de la régulation (évolution de ...).
- Intégration de nouvelles applications ou de systèmes.

À travers les différents travaux de recherches actuelles, nous pouvons affirmer qu'aucun système ne propose à ce jour toutes ces propriétés. Toutefois, les points de flexibilité sur la structure des informations, les règles de coordination et la gestion des exceptions sont de plus en plus présents dans les systèmes dits "flexibles". Ce sont d'ailleurs ces points qui complexifient la coopération inter-systèmes.

4 L'impact de la flexibilité sur la coopération inter-organisationnelle

L'importance de la flexibilité dans les systèmes informatiques d'entreprises est récente. Ceci implique que les mécanismes d'interopérabilité et par conséquent ceux de coopération inter-organisationnelle ne sont peut-être pas adaptés aux structures des systèmes flexibles ou vont même tout simplement à l'encontre du concept de flexibilité. C'est donc deux questions auxquelles il faut répondre :

1) Y a-t-il des propriétés à respecter par les mécanismes d'interopérabilité/coopération par rapport à la contrainte de flexibilité ?

2) Y a-t-il une structure particulière utilisée pour implémenter la flexibilité ? Si oui les mécanismes d'interopérabilité/coopération sont-ils adaptés à celle-ci ?

La seconde question nécessite l'étude des mécanismes d'interopérabilité/coopération existants (chapitre 2) et des structures possibles pour la flexibilité (chapitre 3). Nous pouvons par contre d'ores et déjà apporter une réponse à la première question.

Les paramètres de la coopération entre systèmes sont modifiés principalement par deux propriétés des systèmes flexibles : leur caractère évolutif/dynamique et l'implication des utilisateurs. Les solutions qui permettront de lier de tels systèmes doivent être adaptées à ce nouveau contexte. Nous avons identifié quatre contraintes principales à respecter par ces solutions :

- **l'accessibilité :** les liens entre systèmes doivent pouvoir être définis par les utilisateurs (sinon les systèmes perdent de leur flexibilité). Les moyens d'accès à cette définition doivent être soit très simples, soit réutiliser les outils de "modification" propres à chaque système.
- **la visibilité :** les liens avec l'extérieur doivent être visibles pour chacun des systèmes. Ainsi à chaque modification effectuée par un "utilisateur", celui-ci a conscience des liens avec l'extérieur, et peut veiller à ce que sa modification ne rend pas incohérents les précédents liens ou peut changer ces derniers pour les rendre cohérents.
- **la flexibilité :** les liens doivent bien sûr pouvoir être modifiables eux-aussi (pour ne pas brider la flexibilité des systèmes).
- **un champ d'action plus faible :** on ne doit jamais avoir recours à des langages de programmation (c'est-à-dire aucune intervention des développeurs).

Ces quatre contraintes sont fondamentales. Elles vont guider les travaux présentés dans cet ouvrage. Dans un premier temps, elles nous permettront de démontrer que les solutions actuelles d'interopérabilité ne sont pas adaptées à la coopération de systèmes flexibles. Elles nous guideront ensuite dans l'élaboration de nouveaux mécanismes pour supporter une telle coopération.

5 Origine de nos travaux : l'équipe NOCE

Ces travaux ont été effectués dans l'équipe NOCE (Nouvelles Organisations pour la Coopération et l'Éducation) du laboratoire Trigone. Cette équipe travaille sur le TCAO (Travail Coopératif Assisté par Ordinateur). Les thématiques abordées sont nombreuses : les Interactions Homme-Machine, les collecticiels, les systèmes de workflow, la modélisation et les sciences sociales. La notion de coopération est au cœur de nos travaux (les enjeux d'une coopération, les mécanismes nécessaires, ...) [HOE 01]. Jusqu'à présent, l'intérêt a été porté sur la coopération entre individus. La transition individu → organisation a été motivée par trois raisons :

- la nécessité d'extensibilité des systèmes de TCAO,
- l'étude de la coopération d'un point de vue plus macroscopique,
- notre intérêt pour l'enseignement à distance, grand "consommateur" d'interopérabilité.

5.1 Extensibilité des systèmes de TCAO

Nous nous situons ici dans le cadre des outils qui supportent les activités collaboratives. Dans le cas d'activités évoluées de ce type, plusieurs collecticiels peuvent être utilisés en même temps. Chaque participant doit alors activer au démarrage chacune des applications. Il doit ensuite suivre la stratégie choisie par le groupe pour mener à bien la tâche, notamment respecter les privilèges d'accès qu'ils lui sont conférés : simple spectateur, acteur principal, administrateur... Pour supporter cet aspect, certains outils proposent des dispositifs pour configurer ces privilèges. Malheureusement ces configurations ne sont pas liées entre elles. La cohérence d'un rôle particulier à travers la configuration de ces différents privilèges n'est pas automatiquement validée par un système informatique.

En 1991, l'objectif du laboratoire Trigone, et plus précisément les travaux de F.Hoogstoel étaient de résoudre ces problèmes (activation, cohérence des rôles) en créant un environnement intégrateur de collecticiels [HOO 95]. Le but de cet environnement est que pour une activité donnée, chaque utilisateur voit apparaître sur son ordinateur les outils nécessaires (différents selon le rôle de la personne), et n'a accès qu'aux fonctionnalités qui lui sont octroyées. La structure de l'activité aura été au préalable définie à travers un formalisme de description. Pour avoir une cohérence entre chaque activité, Hoogstoel adopte une approche organisationnelle : le système devient un support au fonctionnement de l'organisation. La définition des activités va passer par la description de la structure de l'organisation où ont lieu ces activités : les activités existantes, les rôles existants, leur présence ou non dans chaque activité. Pour chaque activité il faut ainsi décrire les outils utilisés et les droits d'utilisation pour chaque rôle présent. La proximité avec les systèmes de workflow est ici très présente. Néanmoins, l'attention est portée sur la collaboration et non pas sur la coordination. Les règles de transition et les conditions laissent place aux privilèges d'accès aux outils pour chacun des rôles.

Bourguin amorce ses travaux à la suite de ceux d'Hoogstoel. Il identifie trois problèmes [BOU 00] :

- un problème ontologique : les concepts de modélisation du système n'ont pas la même signification pour les utilisateurs et pour les concepteurs. Ceci perturbe les discussions, entre les précédents individus, qui ont pour objectif de modéliser les activités.
- un problème d'usage : les mécanismes de modification des modèles d'activités sont trop complexes pour les utilisateurs et leur sont donc inaccessibles. Ils sont contraints de définir le plus précisément possible ces modèles dès le départ. Ce qui leur est difficile et diminue de surcroît les perspectives d'adaptation.
- un problème de structure : l'activité n'apparaît qu'à un niveau microscopique et pas macroscopique.

Pour résoudre ces trois problèmes, Bourguin approfondit l'étude des travaux réalisés en sciences humaines sur le concept d'activité en s'inspirant particulièrement de la Théorie de l'Activité (AT) [BOU 00]. Celle-ci a récemment acquis une place importante dans les domaines de l'IHM et du TCAO grâce aux travaux d'Engeström[ENG 87], Kuuti[KUU 91] et Nardi [Nar 96],. Il ressort de ses investigations que l'activité humaine est de nature expansive. Cette propriété est due à l'influence réciproque entre le sujet[11] et le contexte d'une activité :

- le sujet transforme continuellement son contexte de travail (outils utilisés, ordre des actions, ...) pour des raisons d'amélioration, d'automatisation, ...
- Le contexte influence le sujet. Celui-ci doit s'adapter à toute transformation du contexte. Ces changements augmentent l'expérience du sujet et lui permettent d'améliorer le précédent contexte.

L'activité est donc influencée par son contexte qu'elle transforme continuellement. Cette propriété réflexive de l'activité humaine explique pourquoi il est impossible de définir précisément les

[11] Le sujet est une personne ou un groupe de personnes impliquées dans l'activité.

besoins des utilisateurs envers un système informatique car ils apparaissent pendant la réalisation de l'activité où ce système est utilisé. Il est donc logique de disposer des mécanismes de flexibilité de manière à faire évoluer la structure des activités supportées par le système mais aussi de proposer une grande malléabilité pour que l'évolution puisse être effectuée par les utilisateurs. Bourguin conçoit un système, DARE (Distributed Activities in a Reflective Environment), dont l'objectif est de supporter l'activité humaine, en se focalisant plus précisément sur le support des besoins émergents des utilisateurs. L'emphase est mise sur la malléabilité. Pour cela, il adopte une approche composant très adaptée à l'intégration de nouveaux outils qui permet en plus d'encapsuler dans des composants dits "abstraits" des aspects techniques ou des règles de collaboration complexe. Le langage de composition est issu de l'AT pour que la distance sémantique entre les utilisateurs et les concepteurs soient la plus courte.

Mettant en exergue la nécessité du système à pouvoir étendre ses capacités de traitements, Bourguin conclut sur la nécessité qu'ont les systèmes malléables à s'étendre et par conséquent à communiquer avec d'autres systèmes pour utiliser les informations qu'ils contiennent (partage de connaissances), les services qu'ils proposent (comme pour le partenariat) ou pour effectuer une tâche plus rapidement en divisant le travail. Ces conclusions sont à la base des travaux présentés dans ce mémoire.

De façon plus générale, les travaux actuels de l'équipe NOCE porte sur la co-évolution système-tâche [BOU 01].

5.2 La coopération à grande échelle

On peut considérer que toute activité coopérative prend place dans une organisation (aussi minime soit-elle). Et cette organisation entretient nécessairement des liens avec d'autres organisations. Le contexte d'une activité coopérative est influencé par les liens qu'a son organisation avec l'extérieur [BOU 01b]. Leur prise en compte par le système informatique, qui supporte l'activité, ne peut-être que bénéfique : des contraintes ou de nouvelles fonctionnalités peuvent apparaître. Plus le support informatique est global, plus il peut contextualiser une activité et la réaliser avec précision. C'est donc une obligation pour nos recherches sur le TCAO de prendre en compte l'activité dans sa plus grande globalité possible. Si théoriquement, des approches comme l'AT intégraient déjà cette capacité, c'est la barrière technique (l'hétérogénéité des plate-formes) qui empêchait cette vue macroscopique. La flexibilité est une propriété depuis longtemps définie comme indispensable en TCAO. Cette propriété et le fait que la coopération inter-organisationnelle peut être vue comme une simple activité sont les raisons pour lesquelles les contraintes citées en 4 semblent être une re-formulation de la nécessité d'intégration des liens externes que nous venons d'évoquer.

Considérant l'organisation comme élément d'une organisation plus grande, un bon nombre de travaux sur le TCAO peuvent s'appliquer sur la coopération inter-organisationnelle. Par exemple, les six enjeux à atteindre par le support informatique à l'organisation définis par Hoogstoel (cohésion de groupe, améliorer la communication...) [HOE 95] peuvent faire partie des enjeux de la coopération inter-organisationnelle. La mise en relation de l'inter- et de l'intra-organisationnel est bénéfique dans les deux sens. Notre motivation pour amorcer les travaux présentés dans cet ouvrage n'en est que plus grande.

5.3 L'enseignement à distance

L'ACAO (Apprentissage Coopératif Assisté par Ordinateur) est depuis bon nombre d'années une thématique de notre laboratoire [DER 93][DER 94][VIE 98]. Par rapport au cadre plus général du TCAO, l'accent est mis sur la construction du savoir (de l'individu et de la communauté). Nos travaux se sont plus précisément focalisés sur l'enseignement à distance (EAD) et visent à proposer des supports informatiques aux formations à distance ("campus numériques"). Le campus virtuel dont le développement a été amorcé pendant la thèse d'Hoogstoel, puis commercialisé par la société Archimed [ARC 02] est actuellement le support au DESS MICE EAD démarré en novembre 2001.

Les échanges qui ont lieu entre universités ou tout autres centres de formations sont une spécificité du milieu de l'enseignement : un étudiant peut effectuer des modules dans un autre établissement que celui où il prépare son diplôme. Un support informatique à la coopération inter-organisationnelle y est le bienvenu.

Les supports informatiques à ces EAD ont comme nécessité d'être flexibles. Comme tout support à une activité coopérative, ils doivent permettre aux sujets de modifier leur contexte (AT). Nos collaborations avec des pédagogues (équipe MEGADIPE du laboratoire Trigone) nous ont montré que les enseignants souhaitent fortement avoir la capacité d'adapter leur cours aux étudiants.

Nos travaux sur la coopération de groupware flexibles ont donc une grande résonance dans le domaine de l'enseignement à distance. L'exemple pivot présenté dans le chapitre 3 et utilisé dans les chapitres 4 et 5, s'inscrira d'ailleurs dans ce contexte. Il en va de même pour la manière avec laquelle nous abordons la coopération inter-organisationnelle : notre point de vue s'orientera d'avantage vers des coopérations entre systèmes d'EAD plutôt que vers des coopérations entre systèmes industriels où des contraintes vis-à-vis de systèmes légataires ou de la production "matérielle" sont très présentes.

Chapitre 2

Les solutions existantes d'interopérabilité

Dans le chapitre 1, nous avons montré qu'il est stratégique de coopérer pour les systèmes informatiques des entreprises. L'évolution de ces systèmes vers une plus grande flexibilité implique dans cette coopération de nouvelles contraintes (accessibilité, visibilité et flexibilité). L'objectif de ce chapitre est de montrer que les principes de fonctionnement des solutions d'interopérabilité actuelles ne conviennent pas à la coopération de systèmes flexibles. C'est principalement le caractère dynamique des systèmes flexibles et l'implication des utilisateurs qui y ont lieu qui sont à l'origine de cette inadaptation.

Nous commençons d'abord par la comparaison des concepts d'interopérabilité et de coopération. L'utilité de cette comparaison est de reconnaître les solutions d'interopérabilité les plus adaptées à la coopération. Nous étudions ensuite les différents niveaux d'interopérabilité usuellement identifiés afin de mettre en avant la difficulté à construire une solution d'interopérabilité complète. Les critères d'évaluation d'une "solution" sont l'objet de la deuxième partie. Les contraintes identifiées dans le chapitre 1 vis-à-vis de la coopération de systèmes flexibles sont utilisées afin de désigner les critères à privilégier. Dans la troisième et dernière partie, nous présentons les cinq approches élémentaires pour l'interopérabilité. Ces cinq approches regroupent les différentes solutions d'interopérabilité actuelles. Les précédents critères "privilégiés", les contraintes du chapitre 1 et la différence sémantique interopérabilité/coopération nous permettent de démontrer l'inadaptation de ces approches dans un contexte dynamique.

1 Interopérabilité

1.1 Définition

Nous avons indiqué dans le chapitre 1, que la coopération était un cas particulier de l'interopérabilité. Nous souhaitons montrer toutefois que ces deux concepts demeurent fort proches.

Interopérabilité est un terme issu du domaine informatique. Étant très récent, ce terme est rarement présent dans les dictionnaires français ou anglais. Comme pour un grand nombre de termes informatiques (ex : agent), il n'existe pas de définition standard. Il est même rarement défini dans les travaux de recherche. On trouve néanmoins différentes organisations ou sites Internet qui en donnent une définition. Par exemple, le *IEEE Standard Computer Dictionary* [IEE 90] définit simplement l'interopérabilité comme "*the ability of two or more systems or components to exchange information and to use the information that has been exchanged*". Cette définition nous permet de distinguer interopérer et communiquer : communiquer est l'échange d'informations ; interopérer implique une communication et une utilisation de ce qui a été échangé.

Miller, membre du Interoperability Focus, utilise dans [MIL 00] la définition du site WhatIs : "*Interoperability is the ability of a system or a product to work with other systems or products without special effort on the part of the customer. Interoperability becomes a quality of increasing importance for information technology products as the concept that "The network is the computer" becomes a reality. For this reason, the term is widely used in product marketing descriptions.*" Pour Miller, un système qui interopère est un système qui travaille avec un autre système. La distinction avec le terme coopérer, que nous avons employé dans le contexte du partenariat, devient plus difficile à cerner. Il est intéressant de noter que la définition se termine sur la notion de transparence pour l'utilisateur où son ordinateur (ou système informatique) et les "autres" ne font plus qu'un[12]. L'importance de la facilité d'accès pour l'utilisateur aux informations d'autres systèmes conforte notre initiative de coopération où l'implication de l'utilisateur est forte.

Dans le chapitre 1, nous utilisé le terme *coopérer* dans un contexte où les acteurs de la coopération travailler ensemble de leur plein gré et en ayant conscience de la "communauté" ainsi formée. D'après F.Hoogstoel [HOE 95], le terme *collaboration* serait mieux approprié. À la différence de collaboration, la *coopération* n'implique pas nécessairement de relation entre les sujets et la communauté (il peut s'agit d'un simple apport). Néanmoins, il précise que cette distinction est rarement faite, et que coopération et collaboration sont *'utilisés comme des synonymes*". Dans notre cas, nous employons le terme *coopération* avec les implications de la collaboration. Par contre le terme *interopérabilité* n'inclut pas l'obligation de conscience des partenaires. L'exemple le plus marquant est celui des moteurs de recherche existant sur l'Internet : il n'y a pas trace de l'utilisation par *Yahoo!* du moteur de recherche *Google* dans ce dernier[13]. Si *Yahoo!* n'existait pas, l'implémentation du moteur *Google* serait la même (ce qui n'est pas du tout le cas pour le moteur de *Yahoo!*). D'après les deux définitions précédentes, ces deux systèmes interopèrent (informations échangées et utilisées, réutilisation d'informations) mais techniquement, le moteur de recherche n'en a pas "conscience" : le système de *Google* ne peut pas déduire d'une coopération avec *Yahoo!* après une introspection de son code d'exécution. Dans notre contexte, par contre, le fait qu'un système effectue des traitements sur des informations (ou des produits) qu'il reçoit d'un autre système et qu'il renvoie à celui-ci le résultat (l'observation peut se faire pour l'autre système) entraîne une conscience du partenaire : les systèmes peuvent déduire, d'après leur contenu, qu'ils *coopèrent* avec d'autres systèmes.

1.2 Les différents aspects de l'interopérabilité

L' Interoperability Focus [INT 02] définit six types d'interopérabilité : technique, sémantique, politique, inter-communautaire, légale et internationale. A travers cette sous-division, elle décrit les différents problèmes à prendre en compte lorsque deux systèmes interopèrent.

- **L'interopérabilité technique** : il s'agit des mécanismes responsables des échanges de données : la représentation, le transport/communication, la persistance ... Plus la représentation est abstraite, plus elle délimite la nature des données et plus elle facilite l'interopérabilité sémantique.
- **L'interopérabilité sémantique** : Chaque donnée a un sens qui est propre au système dont elle provient. Pour qu'un système puisse utiliser une donnée externe, il doit donc comprendre cette sémantique, c'est-à-dire l'avoir traduite dans son référentiel sémantique (ou avoir le même référentiel). L'interopérabilité sémantique désignent les mécanismes responsables de cette traduction (ou de cette mise en commun) [HEI 95].

[12] Il insiste sur cet objectif d'unicité et encourage la communauté informatique à s'investir dans sa mise en oeuvre : "*one should actively be engaged in the ongoing process of ensuring that the systems, procedures and culture of an organisation are managed in such a way as to maximise opportunities for exchange and re-use of information, whether internally or externally*". Si notre intérêt est d'origine industrielle, pour Miller, l'interopérabilité est une caractéristique inhérente à l'informatique.

[13] Il peut y en avoir, mais elle n'est pas nécessaire.

- **L'interopérabilité politique et humaine** : La stratégie des organisations et la nature de leur effectif doivent être pris en compte. Les relations entre organisations vont décider des privilèges d'accès aux informations. Si les employés n'ont pas les compétences suffisantes pour utiliser un système plus puissant mais aussi plus complexe, l'interopérabilité aura échouée [KLE 96].
- **L'interopérabilité inter-communauté** : Ayant la possibilité d'avoir des informations de plus en plus larges, nous utilisons des sources d'informations de domaines différents. Il est important que les spécialistes de ces domaines se rencontrent pour trouver un maximum de base commune, que chaque communauté n'a pas sa propre technologie. Ce type d'interopérabilité est donc plus macroscopique que les trois précédentes.
- **L'interopérabilité légale** : Des informations peuvent être librement échangées dans certains pays et dans d'autres non. Il en est de même pour les services qui peuvent être autorisés ou non (ex : cryptage de données). Les lois rentrent en compte dans la conception des liens d'interopérabilité [PAE 98].
- **L'interopérabilité internationale** : Les cinq aspects précédents prennent plus d'ampleur dans un contexte international. Les différences de langues, de façons de travailler, de techniques utilisées... amplifient les difficultés.

Lorsque deux systèmes interopèrent, il faut d'abord mettre à plat leur différence de format de données, de transfert ... (interopérabilité technique). Ensuite il faut indiquer les services et données équivalentes et/ou qui utilise quoi (interopérabilité sémantique). Dans cet exercice, il faut prendre en compte les organisations qui utilisent les deux systèmes. Certaines informations doivent être protégées (et ne pas intervenir dans les échanges) ou ne pas être trop évoluées (au risque d'être incomprises et donc inutilisées) (interopérabilité politique et humaine). Ceci peut être complexifié si les deux organisations sont de pays différents où les cultures et/ou les lois sont différentes (respectivement interopérabilités internationale et légale). Enfin, la création de mécanismes d'interopérabilité nécessite la réunion de spécialistes de domaines différents pour qu'ils tissent des liens entre les concepts propres à leur domaine (interopérabilité inter-communauté).

Une solution d'interopérabilité est une aide à la conception de passerelle entre systèmes. Elle accélère la construction de celle-ci. Si un informaticien n'utilise aucune solution particulière, il peut néanmoins créer une telle passerelle. Par contre s'il se sert d'une solution, la construction sera facilitée pour chaque niveau où la solution propose des mécanismes spécifiques[14]. Le coût ou le degré d'autonomie font partie des critères qui permettent à l'informaticien d'évaluer et/ou de choisir une solution par rapport aux autres.

2 Critères d'évaluation d'une solution d'interopérabilité

2.1 Importance d'un système de mesure

Il n'y a pas de solution parfaite pour l'interopérabilité entre systèmes informatiques [PAE 98]. Ce problème reste encore considéré comme très complexe. La complexité provient en grande partie de l'hétérogénéité des choix logiciels et matériels des systèmes participants[15]. Pour évaluer la pertinence d'une solution d'interopérabilité, il est utile d'avoir des critères de mesure. Par exemple, pour la conception d'une application, de tels critères existent : rapidité, flexibilité, universalité, ouverture ...

Cette utilité est aussi présente dans un cadre plus commercial. En effet, les mécanismes qui sont créés pour permettre l'interopérabilité ont des coûts. Ces derniers apparaissent, comme pour une application, à différents moments. Ils sont de plus partagés par les différentes organisations

[14] Il est très rare qu'une solution d'interopérabilité intervienne aux six niveaux précités.

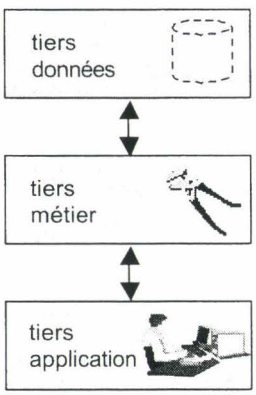
[15] Les problèmes liés aux utilisateurs finaux [STR 96] ne doivent pourtant pas être négligés.

coopérantes. Ceci implique, pour le choix entre plusieurs solutions, la nécessité d'évaluer de manière objective la convenance d'une solution à un contexte donnée. Des critères de mesure pour les solutions d'interopérabilité sont donc là aussi très utiles.

2.2 Mesurer une solution d'interopérabilité

Pour mieux comprendre la portée de ces critères, nous allons voir les caractéristiques structurelles d'un système qui peuvent intervenir dans les mécanismes d'interopérabilité. Nous pouvons décomposer l'architecture d'un système selon le modèle 3-tiers :

1. données : les informations
2. métier : comment manipuler ces informations
3. application : comment les présenter aux utilisateurs



L'architecture OMA définie par l'OMG peut affiner la description des caractéristiques d'un système. Les systèmes actuels disposent en général d'un service de sécurité, de coordination (concurrence d'accès), de transaction, de persistance, de recherche, de description... [SIE 98]. Ces services peuvent intervenir à plusieurs niveaux de l'architecture 3-tiers.

Dans [PAE 98], Paepcke fait le point sur les méthodologies existantes qui peuvent être utilisées dans le contexte des "bibliothèques digitales". Il établit, à cette occasion, une liste de critères d'évaluation des solutions d'interopérabilité :

- degré d'autonomie,
- faible coût de l'infrastructure,
- facilité d'ajout de nouveaux composants ou systèmes,
- facilité d'utilisation des composants ou systèmes, à indiquer dans facilité d'évolution
- degré de complexité de la tâche supportée par la solution,
- possibilité de mise à grande échelle.

Le champ d'application de cette liste dépasse celui des bibliothèques digitales et peut s'appliquer aux systèmes informatiques en général. Nous préférons néanmoins séparer ici les coûts des autres critères. Nous considérons, en effet, les coûts comme l'échelle de mesure de chaque critère.

Dans la suite, le terme "solution" désignera une architecture logicielle et matérielle qui permet de faire interopérer plusieurs systèmes. Le terme "fédération" désignera l'ensemble des systèmes interopérants.

2.2.1 Complexité supportée

Pour mettre en relation deux ou plusieurs systèmes, sont construits des mécanismes d'interopérabilité. Une "solution" ne crée pas tous les mécanismes nécessaires. Par rapport à la situation où des concepteurs devraient faire interopérer des systèmes complètement hétérogènes sans aucune aide, l'objectif d'une "solution" est de faciliter leur travail. Ces solutions peuvent être des standards, des médiateurs... Ce que nous désignons ici par *complexité supportée* c'est à quel point la solution va accélérer la création des liens entre les systèmes. Par rapport à la classification des différentes interopérabilités vue en 1.2, la solution peut aller de l'aspect le plus bas (technique) à l'aspect le plus abstrait (différence de culture). Dans la réalité, les solutions dépassent rarement le niveau "sémantique". Pour évaluer le degré de complexité, on se pose alors les questions suivantes : s'agit-il simplement d'une homogénéisation des transferts de données ? du format de données (format

d'un entier) ? de l'accès aux fonctionnalités (par ex : utilisation du langage IDL dans CORBA) ? de la politique de coordination ? etc.

2.2.2 Degré d'autonomie

Le degré d'autonomie est inversement proportionnel aux contraintes imposées : si une solution tolère un haut degré d'autonomie pour les systèmes qu'elle fait interopérer, cela signifie qu'elle leur impose peu de contraintes. Les contraintes peuvent être d'ordre technique ou sémantique : par exemple, technique lorsqu'il s'agit du format des données échangées et sémantique pour les politiques de coordination choisies (pour les systèmes flexibles).

Vis-à-vis de ce critère, une solution idéale est de supporter tous les types de format de données, de protocoles de communications, de politiques de sécurité, de coordination ... et de permettre à chaque système interopérant d'évoluer comme bon lui semble. Cette solution a pour énorme avantage de ne demander, a priori, aucun investissement de départ à l'entreprise qui souhaite faire interopérer son système informatique. [SUB 00] insiste sur le fait que les investissements de départ constituent un facteur de choix important dans la création ou non d'un partenariat. Ils doivent être inférieurs aux économies réalisées grâce à ce dernier[16].

L'investissement n'est pas forcément nul. Dans le précédent contexte idéal, la solution ne connaît, à tout moment, rien de chaque système. Elle a besoin de découvrir à chaque interaction les caractéristiques des intervenants. Ceci nécessite une description précise (en rapport avec le degré de complexité de la solution) de ces derniers et donc entraîne un coût pour l'entreprise qui fournit le composant. On peut estimer que ces descriptions sont déjà existantes. Il reste alors qu'elles doivent être lues à chaque interaction, ce qui implique un temps d'exécution plus long[17].

Le degré d'autonomie est très similaire à la capacité d'adaptation d'un système : il est souhaitable d'avoir un système très adaptable, mais pas totalement adaptable (montée exponentielle de la complexité de développement[18]).

2.2.3 Facilité d'évolution des systèmes

Lorsqu'un système interopère, il n'est pas pour autant empêché d'évoluer. Un système peut effectuer des changements : ajout, modification ou suppression de données, de fonctionnalités... La suppression (ou la modification) d'un "élément" doit être signalée au reste de la fédération. L'ajout d'un "élément" peut aussi être profitable aux autres systèmes. La question est de savoir quel est le prix de la répercussion d'un changement. Ce prix prend en compte le coût de notification d'un changement et le coût des répercussions sur les autres systèmes.

Nous pouvons illustrer l'importance de ce critère sur l'exemple de la création pour un système d'une nouvelle fonctionnalité. Il s'agit donc ici d'évaluer le coût de cet ajout fonctionnel : coût pour le fournisseur et coût pour les clients. La "solution" de notre exemple est l'utilisation du standard WSDL[19]/SOAP[20]. Le langage WSDL permet de décrire des fonctions accessibles à distance (*web services*). Le protocole d'accès/utilisation de ces fonctions est SOAP. La description des fonctions ou les invocations de celles-ci se font à travers des documents XML. Ce type de solution est

[16] Il est à noter que les investissements à plus long terme (maintenance) sont psychologiquement moins importants dans les décisions de partenariats.

[17] On peut faire une analogie avec l'opposition langages compilés / interprétés

[18] "*We can loosely that the more ambitious a system becomes in considering semantic interoperability, the more flexibility we have in options for interacting with it – and the more difficult it is to implement*" [PAE 98]

[19] WSDL : Web Services Definition Language [CHR 01]

[20] SOAP : Single Object Access Protocol [BOX 00]

généralement utilisée dans le cadre du commerce électronique et se destine à une fédération à grande échelle.

Quand un système dispose d'une nouvelle fonctionnalité, la mise à disposition de celle-ci au reste de la fédération s'effectue de la manière suivante : 1) créer la description du service web (correspondant à la nouvelle fonctionnalité), 2) implémenter l'accès SOAP à ce nouveau service (le code d'implémentation peut être généré par un outil tel que *Apache AXIS* [DIO 02]), 3) enregistrer la description auprès d'un annuaire UDDI[21] (facultatif). Ces trois étapes constituent le coût pour le fournisseur.

Pour compléter l'évaluation du coût de l'évolution du système, il reste à calculer les "dépenses" effectuées par les consommateurs, c'est-à-dire les systèmes "clients" pour accéder à ce nouveau service. Dans les parties métier et/ou présentation de ces systèmes doit apparaître cette nouvelle fonctionnalité. L'implémentation de cette intégration constitue le coût utilisateur.

Il est légitime de penser que la construction de la partie cliente peut être nulle si les systèmes disposent de générateurs d'interfaces utilisateur par rapport à la description WSDL (dans le cas où les services ne sont, bien sûr, pas trop élémentaires). Il ne faut néanmoins pas oublier que le coût d'utilisation est alors reporté sur l'usage laborieux quotidien des utilisateurs finaux : des interfaces générées à partir de description WSDL risquent de ne pas être très explicites. Une solution possible est l'emploi d'un langage qui permet de définir des informations destinées à la génération d'interfaces utilisateur. Une partie des coûts dus à un usage laborieux est alors reportée sur le fournisseur du service qui se doit de créer une nouvelle description "utilisateur" (et sur les systèmes qui se doivent de supporter ce nouveau langage). Cette situation est préférable car le coût fournisseur n'intervient qu'une fois à la différence des utilisations *quotidiennes*. Les coûts d'évolution sont alors plus faibles.

2.2.4 Possibilité de mise à grande échelle

Le dernier critère de mesure est la capacité d'une solution à accepter un grand nombre de systèmes participants. La solution WSDL/SOAP précédente a une haute aptitude à être utilisée à grande échelle : le coût de l'apparition d'un nouveau système participant se réduit aux coûts que celui-ci réalise à déclarer toutes ces fonctionnalités en services web, ... et aux coûts dus à l'implémentation de l'intégration de ces fonctionnalités dans les systèmes clients. Le coût fournisseur diminue avec le succès de la plate-forme utilisée car les opérations de mise à disposition peuvent faire partie de la base de la conception (et ne constitue plus un surcoût). En opposition il y a les solutions ad-hoc. Par exemple, une solution est destinée à faire interopérer deux (ou plus) systèmes spécifiques, agissant tel un médiateur. Elle est constituée essentiellement de traducteurs car les systèmes n'ont pas le même protocole d'échanges. Le principal inconvénient est l'intégration dans la précédente fédération d'un système utilisant un protocole d'échanges que le précédent médiateur ne gère pas. En effet, il n'est pas possible d'exiger du nouveau système participant de supporter un nouveau protocole. C'est une opération coûteuse et elle n'est a priori utile que pour la précédente fédération (si les protocoles utilisés sont peu répandus). C'est au médiateur d'intégrer le protocole du nouveau participant. Malheureusement l'implémentation de cette intégration peut perturber le fonctionnement des systèmes qui interopérer auparavant à cause d'un arrêt momentané du médiateur (nécessaire à la mise en place de la nouvelle version). Les solutions ad-hoc supportent moins bien la mise à grande échelle. Parce que l'ajout de nouveaux "acteurs" y est moins onéreux, ce n'est par contre pas le cas de la première solution, qui se base sur son statut de standard (les participants se mettent d'accord sur une plate-forme d'échange unique).

2.3 Correspondance avec les quatre contraintes de départ

Grâce aux critères précités, nous pouvons affiner notre recherche de solutions adaptées à la coopération de systèmes flexibles. Le critère 'Facilité d'évolution' est bien évidemment celui que nous privilégions le plus. Le caractère dynamique des systèmes flexibles entraîne que la solution doit

[21] UDDI : Universal Description, Discovery and Integration [UDD 02]

supporter le plus facilement et de la manière la moins onéreuse possible tout changement des systèmes.

La 'complexité supportée' est importante aussi pour notre problématique. Nous avons évoqué que les échanges d'informations entre systèmes ont déjà été l'objet de normalisation. L'interopérabilité technique est de mieux en mieux traitée. Par contre, un support informatique à la coopération implique l'automatisation de la coordination. Cela correspond à l'interopérabilité sémantique évoquée précédemment : comment utiliser les informations échangées ? Les solutions à la coopération doivent disposer d'un haut niveau de complexité supportée. L'aspect sémantique est un minimum, les aspects politique et légal sont les bienvenues dans notre contexte organisationnel.

Plus les contraintes apportées par l'emploi d'une solution sont grandes sur les systèmes interopérants, moins la solution est attractive. Le 'degré d'autonomie' est a priori élevé. Néanmoins, il n'y a pas de réelle correspondance avec nos quatre contraintes issues de la flexibilité. Il s'agit plutôt d'une exigence dans l'absolu. Nous avons vu qu'un degré d'autonomie raisonnable est le meilleur choix. L'exigence d'utilisation d'un format spécifique des données demeure acceptable s'il s'agit d'une norme. Par contre, l'emploi d'un format des règles de coordination est plus problématique : les spécificités d'un domaine disparaissent tout comme les 'plus' présents dans les systèmes et absents dans la norme. Les solutions d'EAI supportent généralement une batterie importante de formats de données. Le coût de cette capacité est reporté sur les prix de vente. Au vu de ces prix, on comprend rapidement que ces solutions sont destinées à de grandes entreprises[22]. L'interopérabilité sémantique communément admise comme plus complexe implique que le support d'un grand nombre de formats de coordination, coopération ou tout autres formats de données plus abstraites sera plus difficile et donc plus coûteux pour les concepteurs de ce type de solution. Les prix de ces dernières risquent d'être encore plus élevés, rendant inaccessible aux petites entreprises l'usage de celles-ci.

Les exigences vis-à-vis du critère de 'mise à grande échelle' sont plus subtiles. Le nombre de partenaires d'une société est rarement grand. Plus précisément, le nombre de systèmes avec lequel un système doit coopérer est peu élevé dans notre contexte. Les partenaires économiques de la société Boeing sont très nombreux. D'un autre côté, elle est composée d'une multitude de cellules disposant chacune de leur propre système. Un partenaire coopérant généralement avec une seule cellule, les 'fédérations' sont plus locales et le nombre de participant est peu élevé. La 'mise à l'échelle' est plus stratégique dans le cas de la création d'une bibliothèque numérique : plus le nombre de sources d'informations est grand plus la bibliothèque est importante. L'exemple des annuaires UDDI pourrait néanmoins nous faire penser que la solution doit disposer d'un média de publication qui devient un support à la décision de coopération. Cette perspective où les partenariats se décident à partir de description élémentaire de service, comme celles effectuées en WSDL, nous laisse encore perplexe. Elle est très viable pour des services très légers. Dans notre contexte, par contre, nous envisageons des interactions plus fortes, avec des services plus évolués, où la coopération, conséquente, se décide après discussion entre les partenaires. Il ne s'agit pas de simple service mais d'une véritable synchronisation entre les 'chaînes de productions' de chaque système avec des contraintes temporelles, financières... *communes*. La possible publication de modèles de processus, grâce à la future norme WSFL – *Web Service Flow Language* [LEY 01], est dans ce contexte plus pertinente. Elle ne demeure toutefois qu'un support aux négociations "verbales".

[22] Des solutions comme BizTalk changent cette tendance. Mais elles demeurent généralement moins fournies.

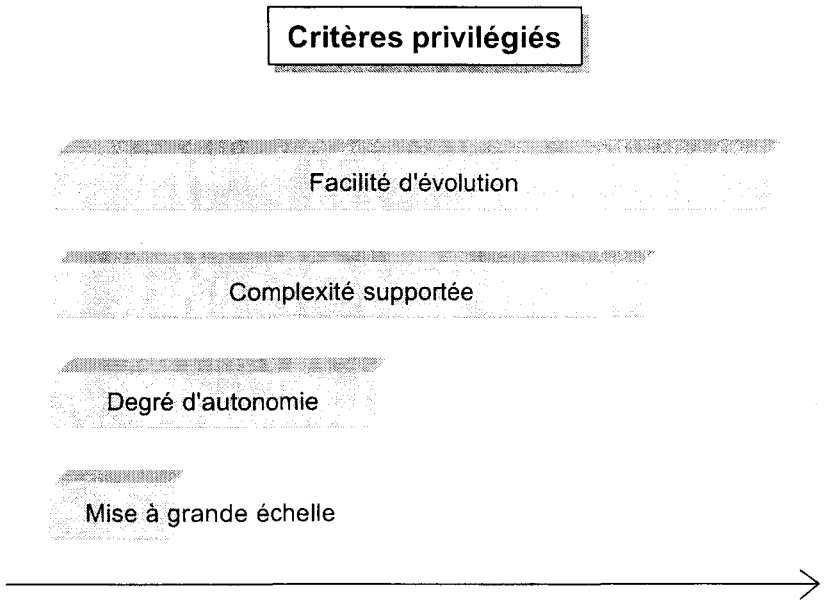


figure 3. L'importance des critères vis-à-vis de notre problématique

3 Les solutions existantes

L'objectif de cette partie est de montrer que les solutions actuelles d'interopérabilité ne sont pas appropriées aux systèmes flexibles. Notre approche n'est pas ici de dresser une liste exhaustive de toutes les solutions existantes (il y en a trop). Nous nous basons sur la classification de celles-ci définie par Paepcke. Il identifie cinq approches élémentaires [PAE 98] : standards forts, famille de standards, médiation externe, interaction par des spécifications et fonctionnalités mobiles. Généralement les solutions réelles sont des combinaisons de ces méthodologies. Chacune de ces dernières a des avantages et des inconvénients. Les solutions réelles utilisent donc ces techniques au vu de leur cahier des charges.

3.1 Les cinq approches élémentaires

3.1.1 Standards forts

Une méthode pour diminuer les problèmes d'hétérogénéité est tout simplement de supprimer l'hétérogénéité : chaque participant utilise le même format d'échange de données, d'informations, de connaissances,... Le protocole TCP/IP, de par sa large utilisation, a permis de supprimer un bon nombre des problèmes de communication. Cette méthode correspond donc à l'utilisation de standard ou de normes. Plus le standard/norme est abstrait, plus un grand nombre de problèmes est résolu. Par exemple, le souhait dans la volonté de placer le middleware CORBA comme une norme était de supprimer les disparités dans les communications, mais aussi dans les échanges de données, d'accès aux fonctionnalités, de notification d'événements, ...

Dans cette approche, la partie délicate est de choisir le bon standard. Paepcke distingue trois types de standards :

- **Les normes convenues** : une grande communauté est d'accord sur le fait qu'un besoin de normalisation pour un problème particulier existe. Elle définit alors une norme. Par exemple, pour permettre la communication entre objets issus d'environnements hétérogènes, un grand nombre d'acteurs du développement logiciel ont créé l'architecture OMA. Ce type de norme n'est pourtant pas forcément très utilisé.

- **Les standards de facto** : si un petit groupe de personnes développe une architecture, un langage etc qui répond à un problème actuel, et que cette réponse est attractive, alors elle peut devenir un standard. L'attractivité est fonction de plusieurs facteurs : le prix, l'utilisation, partenariat avec une grande entreprise...
- **Les normes forcées** : les organisations gouvernementales peuvent parfois intervenir pour soutenir l'évolution d'une norme et, d'une certaine manière contraindre à une acceptation plus répandue (ex : dans le domaine militaire).

Par la suite nous utiliserons indifféremment les termes standard ou norme pour faire référence à l'utilisation d'un format fortement répandu.

Il est donc délicat, vu la variété possible, de choisir un standard suffisamment fort pour que la "solution" soit utilisable par le plus grand nombre. Pour pallier à ce problème, la solution peut alors supporter plusieurs standards.

3.1.2 Famille de standards

Pour fonctionner avec un plus grand nombre de systèmes, une méthode consiste, pour le système qui souhaite interopérer, à intégrer plusieurs standards (d'où l'appellation 'famille'). Pour qu'un système ai ses fonctionnalités le plus facilement utilisable par d'autre, il peut supporter différents accès : à partir des plate-formes .NET [WEI 01], Corba, services web, ... L'utilisation de passerelles telle que *Soap2Corba/Corba2Soap* [SOA 02] permet aux concepteurs du système d'implémenter plus facilement le nombre d'accès supportés.

3.1.3 Médiation externe

L'approche par médiateur externe est apparue au moment où les capacités d'échanges entre systèmes informatiques ont nettement progressé (fibre optique, élargissement des bandes passantes). L'augmentation du volume de données disponibles n'impliquait pas automatiquement une amélioration de la qualité des informations pour, par exemple, l'aide à la décision. Il faut donc mettre en place un intermédiaire, un médiateur pour utiliser au maximum ces données en vue de fournir des informations de meilleure qualité [WIE 92]. Par exemple, l'évolution du prix d'une action d'une société nord-américaine est plus juste, si l'évolution du dollar par rapport au franc est prise en compte.

La médiation s'oriente plus vers une simple coopération qu'une collaboration. Le rôle d'un médiateur est, comme pour le pattern de conception du même nom [GoF 95], d'orchestrer le travail de plusieurs modules/composants et de fournir un composant plus évolué. Dans le contexte d'interopérabilité entre systèmes, un médiateur permet plutôt de composer un nouveau système avec des systèmes ou des sources d'informations déjà existants. Comme elle indique la façon de manipuler les informations, l'approche médiateur est principalement employée pour l'interopérabilité sémantique. Un médiateur peut être un programme spécifique à deux systèmes où les liens d'interopérabilité sont implémentés statiquement. Il peut aussi s'agir d'une plate-forme avec un langage unique de représentation des données. Les systèmes peuvent venir s'y connecter et fournissent dans le précédent langage la représentation de leurs données. Des liens de construction de nouveaux types de données peuvent être définis dans le médiateur. Ce dernier doit pouvoir savoir accéder aux données de chaque système.

3.1.4 Interaction par spécification

L'approche par spécification a pour objectif de donner un maximum de sens aux éléments d'un système pour, par la suite, tisser automatiquement le plus de liens possibles avec d'autres systèmes. Elle nécessite pour cela un effort plus important que les autres approches dans la description des éléments. Cette approche est implicitement employée dans bon nombre d'outils de communication/interopérabilité.

Prenons pour exemple, l'avantage de se servir d'un middleware CORBA par rapport à la simple utilisation du protocole TCP/IP. Deux systèmes (A et B) s'échangent des données en n'utilisant que

TCP/IP. Ces systèmes sont implémentés en utilisant le paradigme objet. Pour les échanges, ils ont besoin de développer leur propre protocole d'échange. Lorsqu'un système envoie une ou plusieurs données, il doit indiquer de quoi il s'agit. Par exemple, dans un contexte bureautique, A envoie à B des renseignements sur un client. Lors de l'envoi il précise au début du transfert qu'il s'agit d'une référence sur une *fiche*. Grâce à cette information, B saura que la chaîne de caractères reçue correspond à la référence d'une fiche. Au système B, ensuite, d'utiliser cette référence pour connaître le nom : il envoie la référence suivie de *nomChamp* (qui indique le champ que l'on désire consulter). Tout ce protocole "objet" qui va être inhérent aux communications (référence, encapsulation de données, vérification de typage, ...) doit être implémenté par les systèmes. L'idée de CORBA est de décrire quels sont les types d'objets présents, et de générer ensuite la partie du code (d'implémentation) d'échange qui gère les mécanismes objet. Une glue avec les objets réels est aussi générée. De cette manière, le système B reçoit l'objet *Fiche* par l'invocation d'une méthode (sur un objet de A) et ne doit plus se connecter au système A, lui envoyer par "TCP/IP" le nom de l'objet, le nom de la méthode, les paramètres ... Il ne doit pas créer de code supplémentaire pour la jonction avec le système A ... mis à part le fait de lui demander une fiche (il s'agit plutôt de fusionner l'ensemble des fiches de A et B). En fournissant la description IDL des types d'objets présents, l'interopérabilité entre les systèmes A et B en est facilitée. Cet exemple illustre le principe de l'interaction par spécification.

La comparaison précédente est utile pour montrer que cette approche est fréquemment utilisée de manière implicite. Les solutions dites "basées sur les spécifications" se situent en général à un niveau plus abstrait. Sur l'exemple précédent, le but ultime serait que *la demande d'une fiche* par le système B à A se fasse automatiquement. Les concepteurs de B n'auraient pas à implémenter une connexion de B vers A et une demande de fiche. Dans les faits, cela se traduit par l'utilisation de langages d'assez haut niveau avec lesquels les concepteurs décrivent la structure de leurs données et de leurs fonctionnalités, la sémantique de cette structure, c'est-à-dire ce qu'elles font, et enfin ce dont elles ont besoin. A l'exécution, la plate-forme (associée aux langages de haut niveau) inspecte chaque description, associée à chacun des systèmes, grâce auxquelles elle va décider des interactions à effectuer. Les interactions entre systèmes ne sont donc pas implémentées au sein des systèmes ni même dans un médiateur, mais déduites et effectuées par la plate-forme à partir des précédentes descriptions.

L'exemple précédent indique qu'il faut bien évidemment que les systèmes coopérants emploient le même langage de haut niveau. Les systèmes multi-agents ou les systèmes à base de connaissance sont des exemples de réalisation de l'approche "interaction par spécifications".

3.1.5 Fonctionnalités mobiles

L'utilisation de la mobilité est une approche différente des autres. Son importance ne se situe pas essentiellement dans le fait de pouvoir faire interopérer des systèmes mais plutôt d'atténuer des problèmes qui interviennent lorsque des systèmes interopèrent. Migrer ou répliquer des fonctionnalités d'un système B vers A permet à ce dernier de pallier à tout arrêt de B. On trouve d'autres avantages [PET 98] comme l'optimisation des coûts de communications ou la répartition de charges.

3.2 Exemples de solutions "réelles"

Nous présentons ici quelques exemples de solutions "réelles" pour concrétiser notre précédent panorama.

3.2.1 Les intergiciels

Les intergiciels[23] sont des exemples de standards : les services web constitue une norme convenue, Corba le fut dans le passé. L'idée commune est que chaque système interopérant utilisant le même format de données. L'intergiciel se charge ensuite de la communication à travers le réseau.

[23] Équivalent français de middleware.

L'intergiciel pourrait être vu comme un médiateur, notamment à cause de l'existence d'un langage pivot. Pourtant ce n'est pas le cas. Il ne contient pas la logique d'interaction. Si par exemple, on pouvait exprimer les interactions dans le langage IDL, on pourrait alors considérer un environnement Corba comme un *mediator facility* (un réceptacle de médiateurs). Si les travaux sur WFDL, langage qui doit permettre de coordonner l'exécution de différents services web, sont concluants, la plate-forme "service web" pourra être considérée comme un *mediator facility* : les interactions entre services web seront définies dans cet élément tiers qu'est la plate-forme.

3.2.2 Data Warehouse (Entrepôt de données)

Le *Data Warehouse* [JON 98] est un domaine où les médiateurs sont légions. Il est aussi au cœur des préoccupations de l'OMG (d'où l'importance de la récente norme CWM [RHO 00]).Cette appellation désigne le fait d'utiliser un ensemble de sources de données (BDR, BDOO, BD-XML, ...) comme composants d'une base de données unique que l'on peut qualifier de virtuelle (BDV). On trouve d'autres appellations [OZS 91] : systèmes de gestion de bases de données multiples, fédération de bases de données hétérogènes, ... Dans ce type de système, l'utilisateur peut accéder à l'ensemble des informations contenues dans les différentes sources à partir d'une seule interface. Par exemple, si une des sources est un SGBDR, et que celle-ci contient la table *Client*, l'utilisateur peut accéder au contenu de cette table. Si la base de données virtuelle est orientée objet, le contenu de la table sera traduit sous forme d'objets. La traduction schéma Relationnel → modèle objet est automatique. Il faut, bien sûr, que la BDV supporte le formalisme relationnel. Le formalisme n'est pas le seul élément à supporter, l'accès informatique l'est aussi (connexion, protocole de communication ...). L'ensemble des modèles objets (issus des traductions) peut servir aussi à créer de nouveaux modèles issus du mixage des premiers. L'aide à la décision (grâce à des informations plus complètes) est souvent un des objectifs recherchés [WIL 98]. Des requêtes OQL peuvent être associées à certains éléments des nouveaux modèles. Les liens entre les modèles de données sont à la charge de l'administrateur de la BDV. Lorsque l'utilisateur effectue une requête, celle-ci est traduite en requête pour les sources de données concernées. En résumé, cette traduction se fait à l'aide de trois choses :

1. La description des modèles de données de chaque source dans le langage de la BDV.
2. Pour chaque modèle précédent, indiquer où se trouvent les données associées, et quel est le type de la source.
3. Des nouveaux modèles de données issus de l'association de modèles existants.

La traduction des requêtes est effectuée par deux types de modules : le médiateur qui, à partir d'une requête envoie les requêtes aux sources concernées et les traducteurs (*wrappers*) qui s'occupent de traduire requête et résultat à partir du langage de la BD virtuelle vers le langage de la source à laquelle ils sont associés. La figure 4 illustre cette description.

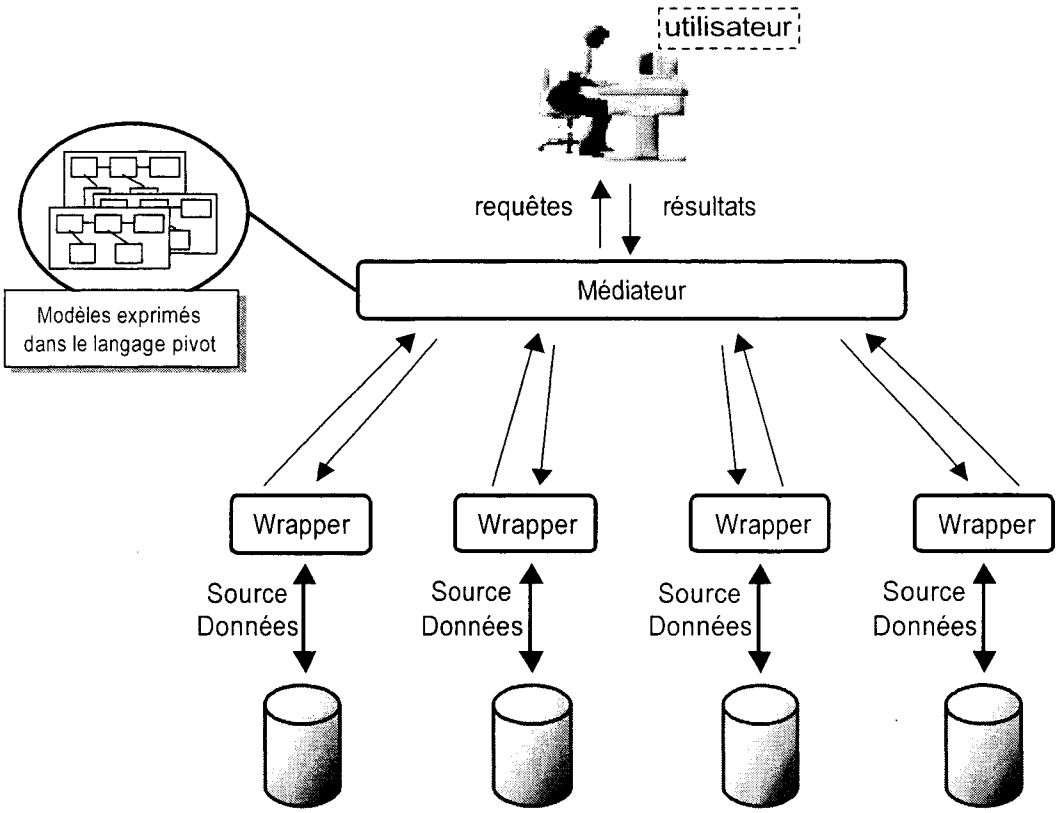


figure 4. Exemple de structure d'un entrepôt de données

Les data warehouses adoptent l'approche médiateur externe. Ce type de solution se repose le plus possible sur des standards : SQL, OQL... (famille de standards).

3.2.3 Les solutions d'EAI

Les solutions d'EAI sont assez proches, d'un point de vue conceptuel, de l'exemple précédent. Il s'agit aussi de médiateurs. Leur objectif est de faire coopérer les systèmes ou applications isolés d'une entreprise. L'origine de l'existence d'îlots informatiques peut être diverse : fusion avec d'autres entreprises, extension vers de nouveaux marchés, acquisition de nouvelles technologies...

Le fonctionnement typique d'une solution est le suivant (cf. figure 5). Le médiateur EAI est d'abord constitué d'un vaste ensemble de **connecteurs** qui lui permette de lire ou écrire des informations dans une application, un logiciel d'ERP, d'invoquer des fonctions dans le contexte d'intergiciels, d'envoyer des messages ou de s'inscrire à des canaux de publications (pour les intergiciels orientés message) ou d'événement (pour les intergiciels comme Corba). À partir de cette connexion, les informations qui doivent être transférées d'une entité informatique à une autre sont transformées du format source au format d'arrivée. C'est la couche **transformations** qui est chargée de cette opération. Les règles de transformation font partie des règles fournies avec la solution (car il s'agit de formats standards), soit elles peuvent être créées par des informaticiens grâce à un environnement fourni à cet effet. Les chemins des informations sont définis dans la couche **routage**. Elle permet d'indiquer selon le type et le contenu des messages le ou les destinataires. Enfin les opérations ou traitements qui doivent être effectués sur ces informations reçues dépendent de l'état du destinataire. Pour gérer ces états, un modèle de coordination est contenu dans la couche **workflow**. Pendant le fonctionnement un état sera associé à chaque entité. L'état variera selon le précédent modèle.

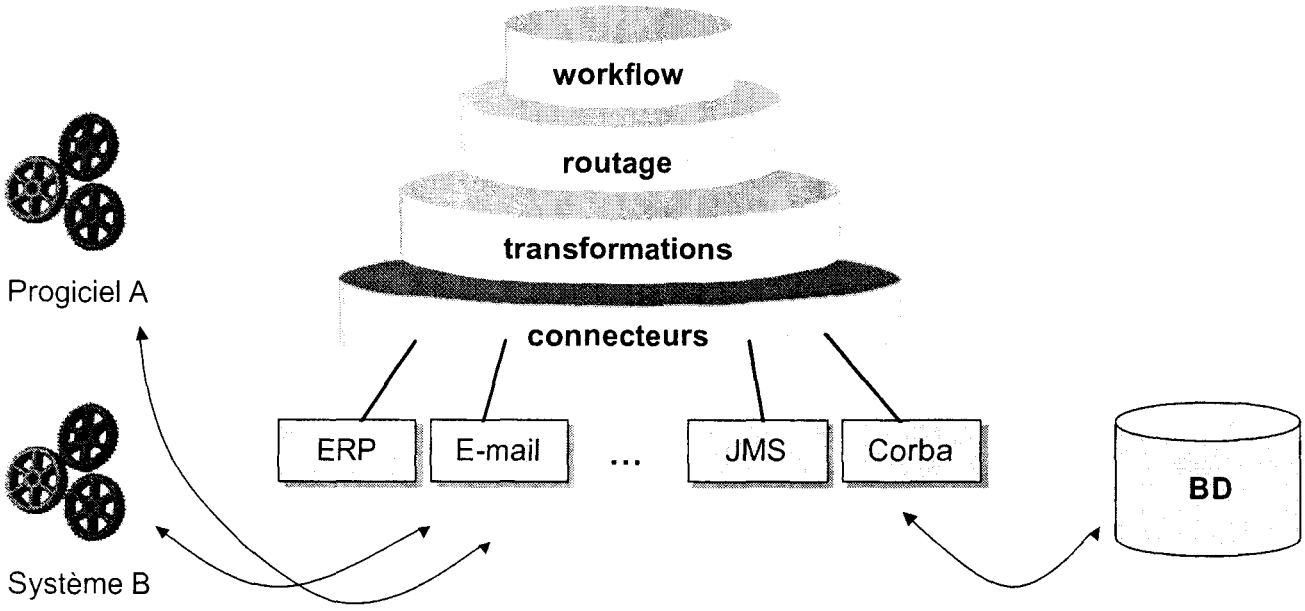


figure 5. Architecture typique d'une solution EAI

À partir de la description des processus (workflow), des règles de routage, de transformations et de l'indication des connecteurs à utiliser (et pour quelle entité l'utiliser), est générée une glue entre chaque système qui reprend la structure décrite dans la figure 5.

Les solutions d'EAI ont été présentées car elles constituent les solutions d'interopérabilité les plus utilisées dans le milieu industriel. Elles sont des médiateurs par excellence et adoptent "massivement" l'approche "famille de standards". C'est d'ailleurs là leur véritable point fort. Ces solutions sont malheureusement très onéreuses.

3.3 Analyse de chacune des approches

Les différentes approches utilisées pour l'interopérabilité présentent des incompatibilités ou ne sont pas viables avec nos préoccupations concernant la coopération de systèmes flexibles. Il faut se rappeler que cette coopération implique des liens entre les types d'informations mais aussi et surtout des liens entre les règles de coordination des organisations.

Standards forts et famille de standards. La première approche demeure la solution ultime : si l'accès aux données, le format des informations, les règles de coordination, la politique de sécurité... sont normalisés/standardisés, alors la construction de la coopération est en bonne partie achevée. Le reste de la conception peut éventuellement consister à implémenter des mécanismes de partage et d'ouverture au sein des systèmes coopérants. Mis à part le degré d'autonomie (les systèmes sont obligés d'utiliser un standard), presque tous les critères d'évaluation sont bien notés. Le degré de complexité est haut car les formalismes de haut niveau sont normalisés. La mise à grande échelle ne pose pas de problème. Seule la facilité d'évolution est quelque peu différente : elle dépend de la généricité des systèmes (accueille-t-il facilement de nouvelles fonctionnalités ?). Malheureusement, au vu des expériences passées, la normalisation est un processus difficilement couronné de succès. Lorsqu'un standard parvient à être défini, il a par la suite une vie mouvementée (ex : apparition fréquente de nouvelles versions). Des produits se disant standards proposent des versions personnelles. Alors des hétérogénéités apparaissent, ce qui diminue l'intérêt de son usage.

Si certaines normes ont réussi à s'imposer, elles touchent généralement un niveau d'abstraction peu élevé : TCP/IP, HTML... La coopération inter-organisationnelle construite à l'aide de standard implique une normalisation des règles de coordination. L'abstraction y est supérieure : le nombre de concepts sous-jacents est plus grand, les dépendances vis-à-vis du système sont moins fortes et les

notions utilisées sont plus proches du langage naturel que d'un langage de programmation. La normalisation risque d'être plus difficile. L'initiative du WfMC à proposer un formalisme standard pour définir des modèles de workflow est souvent saluée. Néanmoins, peu de systèmes "réels" (i.e. pas des prototypes) l'ont adopté. De plus, de nombreux systèmes de workflow sont spécialisés dans des domaines particuliers (cf. chapitre 3) : les modèles de processus sont exprimés dans des formalismes où des spécificités du domaine apparaissent. Ceci complexifie la coopération de deux systèmes agissant dans des domaines différents. Parallèlement à la spécialisation, l'apparition des spécifications EDOC [EDO 01] (Entreprise Distributed Object Computing) et du langage WSFL montre que même la normalisation du format des modèles de workflow (non spécialisés) est loin d'être terminée.

L'utilisation de plusieurs standards se révèle une solution viable pour les échanges d'informations. D'ailleurs n'utiliser aucun standard voue à l'isolement. Par contre, il est très délicat de demander à un système de supporter plusieurs formalismes de description de règles de coordination. La spécialisation des systèmes à des domaines bien précis augmente le nombre de formalismes potentiels à supporter. L'utilisation de standard ne constitue pas une réponse complète à la coopération.

Les médiateurs. Les médiateurs sont des solutions plus "réalistes" que l'utilisation de standard, surtout pour les problèmes d'interopérabilité sémantique. Les médiateurs supportent une grande complexité. Le degré d'autonomie l'est aussi, car c'est au médiateur de supporter les choix architecturaux des systèmes interopérants. À l'inverse de l'usage de norme, il n'y a pas ou peu de contraintes imposées aux systèmes. La facilité d'évolution est plus problématique. En effet, l'évolution d'un système doit être reporté dans le contexte du médiateur et les liens sur la partie modifiée doivent être adaptés. Le surcoût est principalement la transformation dans le langage pivot, la modification des liens étant inévitable. Enfin, la principale incompatibilité s'identifie à l'aide des contraintes de flexibilité : visibilité et accessibilité pour les utilisateurs. Les liens avec l'extérieur n'apparaissent pas dans le système interopérant. Ils n'y sont donc pas modifiables. Si les utilisateurs souhaitent accéder à ces liens, ils doivent apprendre le formalisme pivot utilisé par le médiateur (dans le cas des *mediator facilities*). Ceci est difficilement exigible vis-à-vis des utilisateurs finaux. Enfin dans les solutions d'EAI, la modification d'une entité entraîne la re-génération de l'implémentation de la glue correspondante. Cette opération ne peut être qualifiée de dynamique. Nous verrons aussi dans le chapitre 3, que le passage par un langage pivot peut entraîner des pertes d'informations qui peuvent se révéler dramatiques.

Interaction par spécification. Le degré d'autonomie est la principale qualité de cette méthode. Tout système peut facilement interopérer tant qu'il fournit une description détaillée. L'inconvénient majeur de cette approche est la difficulté d'écriture des descriptions [PAE 98]. À cause de cette difficulté, l'évolution des systèmes est coûteuse. Le deuxième principal inconvénient se situe par rapport à l'implication des utilisateurs. L'ajout d'un élément est accompagné par une description complète de celui-ci. Comme les ajouts sont effectués par des utilisateurs/spécialistes non informaticiens, c'est à eux de fournir la description. Etant donnée la complexité de celle-ci, cela leur est impossible.

Fonctionnalités mobiles. Cette méthodologie a pour inconvénient de nécessiter un support d'exécution commun (non respect du critère d'autonomie) et d'entraîner des problèmes de confiance (d'où certaines politiques de sécurité, comme pour les navigateurs Web et les applets Java). La mobilité n'est pas adaptée à notre problématique. Elle impliquerait qu'un traitement effectué généralement par l'entreprise A (dont c'est la spécialité) soit effectué par l'entreprise B (qui était demandeuse de cette compétence). Ceci ne correspond pas aux relations de type "partenariat" qui nous intéresse.

4 Conclusion

Les méthodologies élémentaires ne sont pas faites pour gérer la flexibilité de systèmes. Seule la standardisation est évidemment LA solution, mais elle est malheureusement irréalisable (elle limiterait en plus la créativité des informaticiens). Si les médiateurs sont plus concevables et attractifs, il demeure le problème de la visibilité de la coopération aux utilisateurs, essentielle pour les systèmes

flexibles. Pour les interactions par spécification, la complexité des descriptions rend cette solution inaccessible aux utilisateurs.

L'implication des utilisateurs et/ou le caractère dynamique des systèmes flexibles ne sont pas pris en compte dans les différents types de solutions existantes. L'importance du support à la coopération de systèmes flexibles a été clairement constatée dans le chapitre 1. Il reste par conséquent à proposer une solution.

Proposer une solution universelle est un objectif utopique car le nombre de systèmes flexibles est assez vaste. Nous devons d'abord choisir sur quel type de systèmes d'entreprise nous allons étudier une possibilité de solution. La sélection peut se faire sur l'indice d'utilisation, sur la présence forte de systèmes flexibles... Ensuite il faut voir quelles sont les réponses techniques existantes pour la flexibilité. Proposer une solution pour tous les types de "flexibilité" existants est difficilement concevable (dans le cadre d'une thèse) car, là encore, il existe différentes approches pour traiter de la flexibilité. Tout ceci est l'objet du chapitre 3.

Chapitre 3

Méta-Modélisation et Technologies de la Collaboration

Nous venons de montrer que les solutions d'interopérabilité actuelles sont insuffisantes pour supporter la coopération de systèmes flexibles. Notre objectif est maintenant de proposer une solution adaptée à ce contexte. Le but de ce chapitre est d'identifier clairement le champ d'application de notre proposition.

Nous l'avons déjà dit dans l'introduction, la coopération inter-organisationnelle peut s'apparenter à une coopération intra-organisationnelle : les règles de coordination des tâches et de collaboration des individus des organisations coopérantes doivent être associées afin de former les règles d'une organisation "virtuelle" union des organisations coopérantes. Les règles de chaque organisation sont généralement gérées par un groupware. Ceci entraîne que la coopération entre organisations se réalise plus naturellement par ce type de système (si les organisations en sont pourvues). Pour cette raison, l'essentiel de notre étude va maintenant se porter sur les groupware – flexibles et les moyens éventuels pour lier ce type de systèmes afin de permettre une coopération inter-organisationnelle.

Après une rapide présentation des groupware (partie 1), nous insisterons sur l'approche la plus efficace pour construire un groupware flexible : l'approche par méta-modèles (partie 2). La description précise de la méta-modélisation et de son opérationnalisation au sein d'un système font l'objet de la partie 3. Un scénario exemple viendra nourrir notre réflexion (partie 4). Cet exemple facilitera la démonstration de l'inadaptation "technique" des solutions d'interopérabilité à l'approche par méta-modèles (partie 5, conclusion de ce chapitre). L'exemple sera aussi utilisé au sein des autres chapitres pour illustrer notre proposition.

Lors de la présentation des groupware, nous évoquons exclusivement les systèmes de workflow (majoritaires dans ce domaine). Toutefois, nous nuançons cette position en insistant sur la spécialisation actuelle de tels systèmes et l'existence d'autres approches pour la gestion du travail de groupe (notamment illustrées dans le scénario exemple). Notre approche doit être suffisamment générique pour englober des systèmes de workflow mais aussi des systèmes comme DARE qui n'ont pas du tout les mêmes bases conceptuelles. *De ce fait, notre étude porte sur le dénominateur commun à ces groupware flexibles : l'approche par méta-modèles.* Elle ne s'attarde pas sur des concepts spécifiques à certains types de systèmes (pour ne pas avoir ensuite une portée limitée).

1 Les groupware

1.1 Notre choix

Le chapitre 1 a retracé rapidement l'évolution du support informatique dans le milieu industriel. Au départ, ce support n'a pour objectif que la gestion 'brute' des données de l'entreprise. Le système informatique ne considère celle-ci que comme un seul élément non composite. Par la suite, l'informatique se présente comme support au travail individuel des employés. Cette gestion des

activités qui ont lieu au sein de l'entreprise devient complète avec la prise en charge de la coordination/coopération/communication de ces activités individuelles par le système informatique. Si cette prise en charge apparaissait déjà dans certains SGBD évolués ou dans les systèmes de GED (Gestion Électronique de Documents), elle a été clairement identifiée comme telle avec l'arrivée des systèmes de workflow (WfMS). Ces systèmes occupent une place de plus en plus importante dans les systèmes industriels [MOO 00]. Ils finalisent ainsi le support informatique à l'organisation : un tel support effectue ou propose une assistance aux différents traitements nécessaires sur les informations, traitements qu'il coordonne et pour lesquels il accomplit les échanges d'informations entre eux nécessaires. La gestion des données y est complètement intégrée. La coopération entre entreprises (ou entre organisations) va correspondre à la coopération de ce type de systèmes.

Pourtant, nos travaux ne se focalisent pas que sur les systèmes de workflow. La principale raison a déjà évoquée dans le chapitre 1 au sujet de la normalisation/standardisation comme solution d'interopérabilité : des systèmes dérivés des WfMS apparaissent ou vont apparaître (cf. figure 6)... Nous y évoquions l'enseignement à distance et les campus numériques comme exemples de spécialisation des systèmes de workflow car il s'agit de notre domaine de recherche. Le système COW -*Cooperative Open Workflow*- (étudié plus loin) est un WfMS où l'orientation apprentissage coopératif vient influencer le noyau fonctionnel. D'autres spécialisations de ce type existent. Les sociétés HSP [HSP 02] ou Elliot (avec Solution1Alliance [SOL 02]) fournissent des WfMS spécialisés dans les activités médicales (suivi de patients, ...). Les applications de gestion d'Autodesk sont construites autour d'un WfMS dédié à l'industrie lourde comme l'automobile [AUT 02]. Le commerce électronique a lui aussi sa place dans cette spécialisation à travers des travaux comme ceux de N. Karacapilidis [KAR 01]. Les assurances ou les sociétés bancaires sont aussi clientes de cette spécialisation (ex : FINEOS [FIN 02]). Même le calcul scientifique dispose avec WASA d'un WfMS où le formalisme de modélisation est dédié à ce domaine.

Les WfMS "généraux" vont bien sûr continuer à évoluer mais des changements notables risquent d'apparaître très prochainement. Récemment, la recherche de flexibilité a déjà provoqué bon nombre de changements dans la structure et l'utilisation de ces systèmes (cf. partie 2). L'intégration prochaine d'aspects collaboratifs comme la notion de droits de partage d'un outil (issue des collecticiels) risque aussi d'apporter des modifications importantes au sein des WfMS.

Dans le domaine des groupware, les WfMS en constituent la partie la plus importante (efficacité, dominance de marché). Car nous nous intéressons à étudier les solutions de coopération de groupware ou à en proposer une, les WfMS vont être le centre de notre étude (et de notre proposition). Toutefois nous n'étudions que leurs principes de base car ces systèmes sont de plus en plus spécialisés. Le reste de cette partie dresse une liste des différentes versions existantes et décrit les concepts de bases inhérents à ces systèmes.

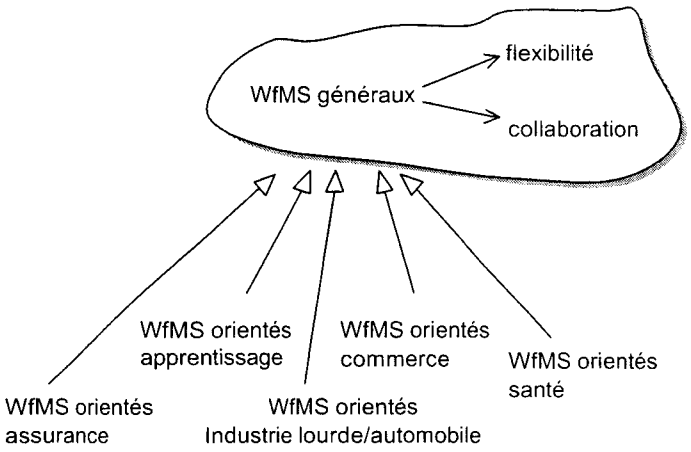


figure 6. Grande mouvance des WfMS : évolution et spécialisation

Il est très important de noter que des systèmes adoptent d'autres approches, différentes de celle des WfMS, pour la gestion du travail de groupe (ils restent toutefois moins nombreux). Le système DARE, longuement décrit dans le scénario exemple (cf. 4.1), en fournit une belle illustration.

1.2 Principe des systèmes de workflow

L'objectif d'un WfMS est d'abord de gérer des processus correspondants aux activités à réaliser. Dans un premier temps, des spécialistes (responsables du management) définissent un modèle des processus de l'organisation. Ces modèles sont ensuite traduits en instructions informatiques exécutables. Les services proposés (fonctionnalités présentes) lors du fonctionnement du WfMS sont nombreux. Comme nous l'avons déjà dit, son principal service est de coordonner les différentes activités d'un processus et de synchroniser l'intervention des différents acteurs. Il permet aussi de suivre l'évolution d'un processus et de garder une trace de l'historique de celui-ci. Pour une activité, il donne à ou aux acteurs un accès à l'ensemble des informations propres au processus pour que celui ou ceux-ci aient une meilleure contextualisation de l'activité. Enfin le WfMS gère la liste de tâches (équivalent à "activités") à effectuer pour chaque processus et une liste de tâches pour chaque participant (avec les applications et informations associées à chacune d'elles). *Un WfMS définit, gère et exécute des procédures en exécutant des programmes dont l'ordre d'exécution est prédéfini dans une représentation informatique de la logique des procédures – les workflows* [WFM 99].

Les WfMS peuvent se regrouper principalement en trois catégories [ZUR 00] : ceux qui coordonnent et utilisent les autres systèmes ou applications de l'entreprise (ex : Endeavors [KAM 98]), ceux qui ne sont qu'un module parmi d'autres dans des progiciels de gestion interpée – ERP – (comme certains précédents exemples de spécialisation), et enfin qui font partie d'une suite logicielle plus large (ex : *MQSeries WorkFlow* d'IBM lié à *Websphere* [MQS 02]).

Les choix de formalisme de définition des processus ou de structure d'implémentation sont très variés. Il y a toutefois des concepts et des éléments structurels communs.

1.3 Concepts de base.

Même si un grand nombre de formalismes de modélisation workflow existent (Réseaux de Petri [VAN 97], SADT, ou autres [CAR 97]), un certain nombre de concepts sont inhérents à tous les WfMS. On peut se baser sur les concepts définis par le WfMC pour en faire l'inventaire (cf. figure 7).

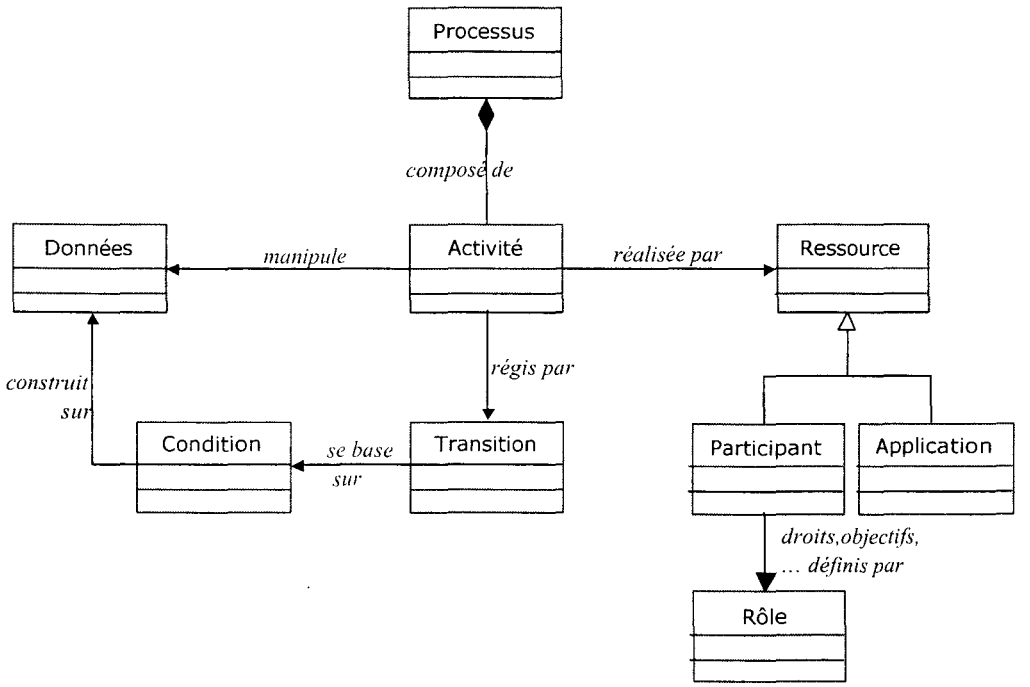


figure 7. Modélisation UML "simplifiée" des concepts workflow du WfMC (ajout du concept de rôle)

L'organisation est le siège d'exécution d'un ensemble de **processus**. Un processus peut correspondre à ce que nous avons appelé dans le chapitre 1 un "procédé métier". Il peut aussi s'agir d'un sous-processus d'un procédé métier. Chaque processus est composé d'**activités**. Il s'agit d'étapes élémentaires (non composites) d'un processus. C'est généralement un travail individuel (ou tout au moins pour l'instant). C'est là les deux principales différences entre l'activité et le processus (qui lui est composite et peut concerner plusieurs acteurs). Remplir un formulaire de bon d'achat et rédiger un rapport sont des exemples d'activité. Le terme tâche est équivalent à activité[24]. Une activité est réalisée en mobilisant des **ressources**. Elles peuvent être humaines (**Participant**) ou informatiques/matérielles (**Application**). Le participant réalise une activité à l'aide d'une ou plusieurs applications informatiques. Cependant, le participant demeure pour le système une ressource (un exécuteur). La gestion des rôles et des compétences est évoquée mais pas définie précisément dans l'ensemble des concepts (standards) du WfMC. Le **rôle** est un concept très présent dans les WfMS. S'il n'est pas clairement défini dans le méta-modèle de WfMC c'est sûrement car 1) les divergences vis-à-vis de ce concept sont très marquées 2) l'importance du concept de rôle dans les échanges entres WfMS (objectif du méta-modèle du WfMC) n'est peut-être pas évidente. Le rôle indique quelles sont les compétences attribuées à un acteur dans le contexte d'une activité ou d'un processus, quels sont ses devoirs... La littérature est très prolifique à ce sujet (d'où les divergences) [LEI 97]. Les **données** sont le support de réalisation d'une tâche. En général, à partir de données dites "entrantes", une ou des nouvelles données sont créées ou des données déjà existantes sont modifiées. Le démarrage d'une activité est régi par un ensemble de règles de **transitions**. Chaque transition part d'une ou plusieurs activités et en déclenche une ou plusieurs autres. Une transition se base généralement sur une **condition**. Elle peut être écrite dans différents langages (choix des concepteurs du système).

[24] Alors que dans les travaux en IHM, notamment menés au sein du laboratoire TRIGONE, il y a une distinction nette entre Activité et Tâche. Ceci sera précisé lors de l'étude du système DARE dans ce chapitre [BOU 01].

1.4 Architecture générale

L'architecture générale d'un WfMS est essentiellement composée de trois éléments (cf. figure 7) :

- Un éditeur de modèles : il est assez souvent graphique. L'éditeur a donc pour premier objectif de permettre la définition de modèles de processus. Son deuxième objectif est de traduire ces modèles en code "informatique" adapté à l'environnement d'exécution. Certains WfMS ne proposent pas d'éditeurs.
- L'environnement d'exécution : c'est la partie la plus complexe et la plus conséquente du WFMS. Il gère l'exécution des processus et des activités de l'organisation. Elle distribue les tâches aux participants concernés en respectant la chronologie/ordonnancement indiqué dans les modèles précédents. Il rend accessible à chaque acteur l'ensemble des données et des outils nécessaires. Enfin il est chargé de vérifier le bon fonctionnement (cohérence) des activités. L'élément principal de l'environnement d'exécution est le "moteur" d'exécution (*enaction*). Y est associé un gestionnaire de tâches qui attribue les activités en fonction des rôles, des disponibilités des participants, des ordres de priorités... Des outils d'administration et de monitoring sont aussi attachés au moteur.
- Les accessoires : il s'agit des outils "bureautiques" de base fournis avec le WfMS. On trouve aussi des connecteurs pour les applications extérieures supportées par le système[25]. Enfin des outils de personnalisation (la plupart du temps pour les interfaces utilisateurs) sont parfois présents.

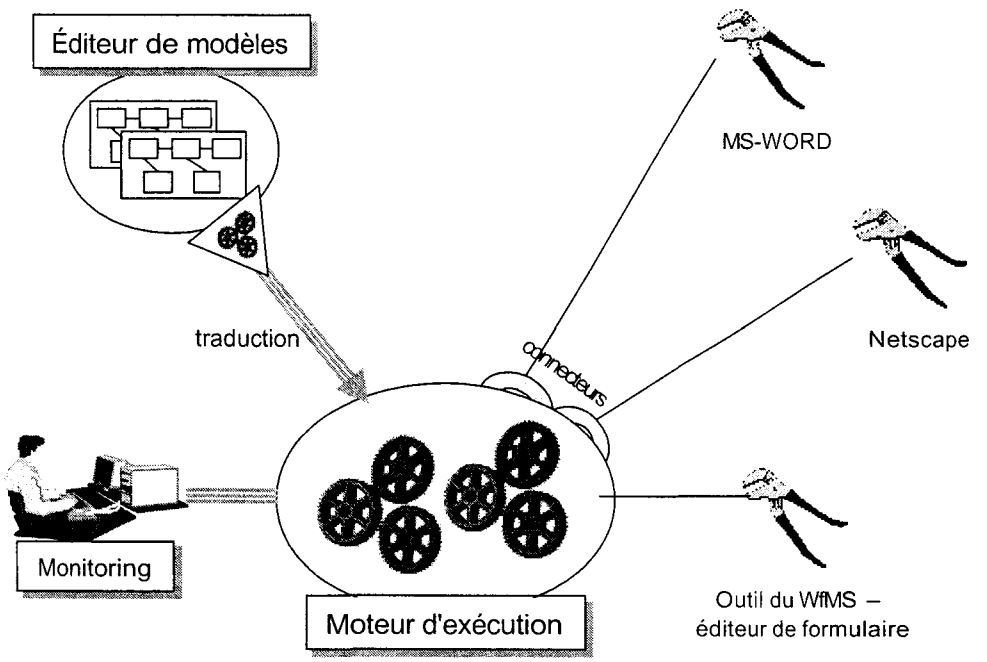


figure 8. Architecture général des WfMS

Cette architecture commune à tous les WfMS constitue les fondements des futurs supports informatiques aux organisations et donc la base de nos travaux. Déjà, il nous est possible d'affirmer que la coopération entre entreprises nécessitera de tisser des liens entre les modèles de processus de chaque système interopérant. Nous allons maintenant décrire plus en détail les techniques pour apporter de la flexibilité à ces systèmes.

[25] Il s'agit du même type de connecteurs décrit pour les solutions d'EAI. Dans ces dernières la panoplie de connecteurs proposés est plus large (c'est d'ailleurs leur principale qualité).

2 La flexibilité des WfMS

Dans cette partie, sont présentées les différentes approches vis-à-vis de la flexibilité des WfMS. Pour cette présentation, nous nous sommes basés sur les travaux de Y. Han [HAN 98]. Nous évoquons ensuite l'analyse de Morch dont la portée, plus générale, n'est pas limitée au domaine du workflow. Ceci rejoint nos préoccupations à ne pas nous focaliser sur ce type de système. Les études de Han et Morch étant au final assez proches, il nous sera permis d'affirmer que le choix de la structure sur laquelle nous allons baser notre proposition n'est pas fait à partir de techniques propres au WfMS mais aux groupware en général.

2.1 Les différentes approches

Deux méthodes principales existent pour intégrer des mécanismes de flexibilité dans une WfMS : les approches ponctuelles et les approches par méta-modèles.

2.1.1 Approche ponctuelle (*open-point approach*)

Les systèmes de workflow contiennent généralement plusieurs modèles de processus qu'ils exécutent. L'approche ponctuelle consiste à placer à y placer des blancs ou des paramètres pour permettre à l'utilisateur d'intervenir, lors de l'exécution, dans les choix de coordination ou d'affection. Trois techniques se distinguent dans cette approche.

Les choix multiples. Les éléments insérés ici dans les modèles sont appelés des bornes. Ces dernières contiennent différentes valeurs, type d'activités, de ressources, parmi lesquelles l'utilisateur doit faire son choix. Par exemple, lorsque le moteur d'exécution arrive sur une borne, l'utilisateur choisit librement une activité parmi une liste proposée (*To do list*).

Allocation dynamique de ressources. Cette technique concerne le problème de répartition et régulation des ressources énoncé dans le chapitre 1 au sujet des points de flexibilité. L'objectif de cette technique est de régler les conflits d'attribution des ressources matérielles ou humaines. Dans les modèles de processus, il est généralement obligatoire d'indiquer quelle ressource est utilisée pour l'accomplissement d'une tâche. A l'exécution, des problèmes peuvent survenir si la ressource n'est pas disponible ou si elle est requise simultanément par plusieurs activités. La première solution est d'utiliser un programme d'ordonnancement qui permet de mieux répartir l'utilisation de la ressource. Dans la deuxième solution, les concepteurs ou/et les "spécialistes" (i.e. responsables du management ou *business analysts*) indiquent dans les modèles de processus les types de ressources utilisées (et non pas directement les ressources). Au moment de l'exécution de la tâche, il peut toujours y avoir le choix entre plusieurs ressources. Est présent dans le WfMS un service de répartition qui s'occupe de sélectionner la meilleure ressource (disponibilité). Lors de ce choix, l'utilisateur peut être sollicité.

Modélisation tardive. La modélisation tardive est une évolution de l'allocation dynamique : en plus de pouvoir placer des types de ressources au sein des modèles, on peut aussi y placer des types de processus, d'informations... Ces éléments génériques sont aussi appelés *templates*. Le modèle est ensuite spécialisé à l'exécution. La limite est assez floue entre certaines versions de modélisation tardive (très évoluées) et l'approche par méta-modèle.

2.1.2 Approche par méta-modèles (*meta-model approach*)

Pour présenter l'approche par méta-modèles, un rappel sur la notion de méta-modèle est à apporter. La méta-modélisation sera réellement approfondie dans la partie 3.

2.1.2.1 Méta-modèles

Modéliser consiste à décrire un modèle. Il est constitué de types de données et de relations entre ces types. Ce modèle est construit à partir d'un méta-modèle. UML [MUL 97] comme les langages de programmation sont des exemples de méta-modèles. Les éléments d'un méta-modèle sont appelés entités. Les concepts de classe et d'attribut Java sont des exemples d'entités. Les entités du méta-

modèle du WfMC sont *ProcessModel*, *Process*, *Activity*, *Resource*... Elles permettent, par exemple, de décrire le traitement d'un achat (modèle de processus *TraiterAchat*) qui contient entre autres, le modèle d'activité *RemplirBonReception*. Un achat particulier comme l'achat d'un Dell XPS est ensuite géré par une instance de *TraiterAchat*. Par la suite, l'expression *structuration en trois niveaux* fera référence au trio méta-modèle/modèles/instances de modèles.

Le méta-modèle n'est pas le seul élément de cette approche. Les éditeurs de modèles contiennent, explicitement ou implicitement, un méta-modèle. Pourtant cela n'indique pas forcément que le WfMS associé utilise l'approche par méta-modèles et qu'il est flexible. C'est l'utilisation de mécanismes de réflexivité, deuxième principal élément de cette approche, qui en est la preuve.

Le fait que les utilisateurs (plus précisément les *spécialistes*) puissent définir leur propre modèle de processus, c'est-à-dire adapter le WfMS à leurs propres habitudes de travail, est déjà une élément de flexibilité. Le réel avantage des approches par méta-modèles est de permettre la modification des modèles alors que des instances de ceux-ci existent. Diverses questions surviennent alors : si on change le modèle de processus *TraiterAchat*, qu'advient-il de l'achat du Dell XPS qui est en cours ? Si des modifications sont faites sur la structure d'un bon de commande, le bon du Dell XPS est-il modifié ? Dans le contexte des approches par méta-modèles, la coordination de l'achat du Dell XPS et son bon de commande subissent les répercussions des changements et sont donc modifiés en conséquence[26]. Dans ce contexte, les modèles définis à l'aide de l'éditeur ne sont pas traduits en termes informatiques : ces modèles existent tels quels dans le WfMS et sont modifiables à partir de l'éditeur. La modification dynamique est permise grâce à des mécanismes de réflexivité.

2.1.2.2 La réflexivité

La réflexivité ou plutôt la réflexivité informatique (*computational reflection*) est un terme issu de la programmation. Dans [BOB 93], la réflexivité est définie comme la capacité d'un système (d'un programme) à manipuler quelque chose, comme des données, représentant l'état du système pendant son exécution. On peut alors différencier dans le système deux niveaux : le niveau de base (ou niveau applicatif, fonctionnel) et le méta-niveau. Le niveau de base correspond à ce que l'on appelle communément niveau applicatif ou niveau fonctionnel. Le méta-niveau, c'est-à-dire le niveau supplémentaire par rapport aux systèmes non réflexifs, contient la description du niveau de base. L'opération qui consiste à traduire la description du niveau de base en termes informatique (le *quelque chose*) s'appelle la *réification* [MAE 87]. À partir de ce méta-niveau, peuvent s'exécuter deux types d'opérations : celles se rapportant à l'intercession et celles à l'introspection.

L'introspection consiste pour un système à la faculté de pouvoir s'observer et ainsi à raisonner sur son propre état. L'intercession, quant à elle, réfère à la capacité du système à modifier son état (pendant son exécution). Techniquement l'introspection et l'intercession concernent quatre aspects du système (entre parenthèses se trouvent le termes d'origine [BRO 93] destinés à un programme) :

1) L'aspect structurel (*structure*) : le modèle de fonctionnement (ex : un modèle de processus).
2) L'aspect comportemental (*program*) : comment la structure précédente réalise le comportement spécifié (ex : le séquencement de sous-activités).
3) L'aspect interprétatif (*process*) : les mécanismes d'interprétation de la structure et de son comportement associé (qui ont pour objectif de produire du code exécutable).
4) L'aspect extensif (*development*) : comment modifier les précédents mécanismes.

2.1.2.3 Les WfMS réflexifs

Proposer l'introspection et l'intercession pour les deux derniers aspects est essentiellement l'adage des langages orientés objet réflexifs comme CLOS [KIC 91] ou 3-KRS [MAE 87]. Les WfMS

[26] Il demeure toutefois une politique de "modification" qui, selon l'état de l'instance, décide s'il y a répercussion ou non. Chaque WfMS "méta" a sa propre politique.

réflexifs se limitent aux deux premiers aspects tant au niveau introspection qu'au niveau intercession. Un WfMS qui dispose de l'introspection permet à l'utilisateur de connaître le modèle dont est issue une tâche en cours d'exécution. Le système peut aussi se servir de cette capacité à ses propres fins dans un contexte d'adaptabilité : une partie d'un WfMS peut analyser les modèles de processus qu'il contient (saisis par les spécialistes) pour supprimer des incohérences, des éléments redondants, l'analyse peut être portée sur les réalisations de ces modèles en vue d'optimiser ces derniers (suppression de ressources inutilisées, d'activités trop coûteuses...). L'intercession pour les WfMS correspond à la capacité de modifier les modèles de processus même quand des instances de ces modèles existent déjà. Il s'agit par exemple de remplacer un type d'activité par un autre au sein d'un modèle de processus, ou de supprimer une activité (jugée inutile), de modifier la condition d'une transition, les compétences d'un rôle...

Les modifications qui sont problématiques sont celles qui remettent en question la nature ou l'existence des instances. C'est le point où les stratégies diffèrent pour chaque système. Remplacer, dans un modèle de processus, un type d'activité par un autre n'est pas gênant si aucune activité de "l'ancien" type n'existe. Cela ne gêne pas non plus les instances du modèle. Par contre, si des activités de ce type existent, que faut-il faire ? Arrêter chacune des activités en cours et les remplacer par une du "nouveau" type ? Laisser terminer ces activités... mais dans ce cas, les transitions associées à ce type d'activités sont-elles conservées de manière à permettre aux processus concernés (i.e. pour lesquels il y a une activité de l'ancien type en cours) de fonctionner sur l'ancien modèle ? Dans ce cas, faut-il effectuer un calcul pour savoir quels éléments conserver (transitions, types de rôles, autres types d'activités, etc.) de "l'ancien" modèle ? ... Les réponses à ces questions peuvent varier selon les systèmes. Elles font partie de la stratégie d'application de l'intercession.

2.1.2.4 Implémentation de la structuration en trois niveaux

Les WfMS (réflexifs) adoptant l'approche par méta-modèle différent par leur politique de répercussion (des changements au niveau des modèles). La principale différence entre ce type de WfMS est le méta-modèle. Jusqu'à maintenant chaque WfMS de ce type a un méta-modèle différent. **La spécialisation des systèmes workflow ne va pas diminuer cette tendance.** L'autre différence majeure est l'implémentation de la structuration en trois niveaux. Pour un méta-modèle, on peut trouver plusieurs implémentations possibles. Quatre choix sont à l'origine des différences :

- Le paradigme utilisé : dans quel paradigme informatique (relationnel, objet, réseau de pétri, agent, ...) est traduit le méta-modèle "abstrait" ? En d'autres termes, dans quel paradigme sont décrits les modèles et leurs instances ?
- Le support informatique : il s'agit du langage (Java, C++, VB) ou de la plate-forme utilisée (EJB, .NET, Corba, WSDL/SOAP, ...).
- Le patron de conception : de quelle manière sont utilisées les possibilités du support informatique et quelle est la structure des éléments représentant les modèles (et leurs instances) ?
- La convention de nommage : la traduction de caractéristiques de concept (du méta-modèle "abstrait") traduit dans le support informatique peut parfois nécessiter l'utilisation d'une convention de nommage.

L'exemple suivant illustre cette distinction. On décide de concevoir un WfMS autour du méta-modèle simplifié de la figure 7. Nous décrivons précisément les concepts du méta-modèle dans le paradigme objet (choix 1). Le support d'implémentation est le langage Java (choix 2). Son module RMI nous permettra la gestion de la distribution. Pour créer dynamiquement de nouveaux modèles, on peut employer différentes techniques de programmation [MAN 98b] : la spécialisation de classes (ex : 2Flow [SAI 01]), le patron de conception type-objet [JOH 95] (ex : Endeavor [KAM 98]), ou même une approche par prototype[27] [TIC 97] (choix 3). Nous choisissons le patron de conception type-objet

[27] Malheureusement peu adapté à l'approche par méta-modèle à cause de l'inexistence d'un type explicite.

(évoqué plus loin) : sont définies les classes *ProcessusType*, *Processus*, *ActiviteType*, *Activite*... La valeur d'une propriété est gérée par un ensemble d'opérations (et non par un attribut). Nous préfixons ces opérations par get, set, add... selon leur action (choix 4).

La coopération entre des WfMS adoptant l'approche par méta-modèles ne se résume pas à la résolution des différences de méta-modèles. Elle doit aussi prendre en compte les quatre types de différences précités.

2.1.3 Apport de l'orienté objet

Sur la base des travaux de Bussler [BUS 98], Saikali [SAI 01] insiste sur l'apport du paradigme objet dans le domaine du workflow. Le concept d'objet est parfois utilisé comme tel dans le méta-modèle utilisé par le WfMS. À d'autres moments (cas plus fréquent), ce sont les caractéristiques de ce concept (l'encapsulation, l'héritage et le polymorphisme) qui sont associées aux concepts du méta-modèle workflow. Par exemple, un type de processus peut hériter d'un autre type. Ces propriétés sont surtout importantes dans le cadre de la réutilisation, caractéristique très importante pour la flexibilité : permettre aux utilisateurs de définir un modèle en se basant sur un modèle déjà existant plutôt que de partir de zéro.

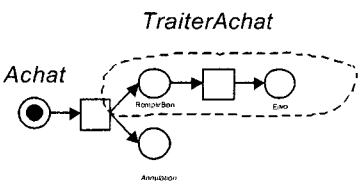
L'héritage permet par exemple d'utiliser un modèle de processus en ayant pourtant spécialisé certains éléments : les autres éléments ne nécessitent pas de modifications (sauf en cas d'incohérence ou d'utilisation des nouvelles caractéristiques).

Exemple d'héritage sur TraiterAchat

> Un modèle complet existe où s'intègre le type de processus *TraiterAchat*. On souhaite modifier l'élément *TraiterAchat* de ce modèle tout en conservant cet élément (si jamais la modification se révèle inutile, incohérente, ...). On désire aussi ne pas modifier les éléments du modèle qui "s'adressent" à *TraiterAchat*. La communication que ces éléments avaient avec un processus de type *TraiterAchat* peut se faire avec un nouveau type de processus si celui-ci a la "même forme" que *TraiterAchat*. C'est ce que permet l'héritage (d'où le terme *polymorphisme*). On fait simplement hériter l'élément remplaçant *TraiterAchat* de *TraiterAchat*. L'héritage permet une réutilisation rapide d'un modèle : modification d'éléments sans avoir de répercussion sur les autres éléments en rapport avec ceux modifiés (tant que les éléments "inchangés" n'utilisent pas explicitement les nouvelles caractéristiques des éléments "modifiés").

L'encapsulation est le mécanisme pivot de la modularité. Elle permet d'associer explicitement un comportement à un type de structure. Elle permet aussi de cacher des éléments complexes. L'utilisateur lambda peut manipuler un type de processus très complexe, par exemple en l'associant à un autre type de processus plus "léger" (défini par l'utilisateur), sans pour autant comprendre la complexité qu'il encapsule. Ce type n'est qu'un "objet" comme un autre.

Exemple d'encapsulation sur TraiterAchat

> *TraiterAchat* peut encapsuler des tâches administratives complexes. Un spécialiste qui n'est pas à l'origine de *TraiterAchat* peut pourtant le placer dans un autre modèle sans en connaître précisément le contenu. D'après la partie visible (ex : nom, rapide descriptif,...) il sait que ce type de processus lui convient. C'est l'encapsulation qui permet de simplifier des éléments et leur manipulation. Sur un réseau de Pétri (ex : [VAN 97]), *TraiterAchat* n'est que la délimitation d'un ensemble de nœuds interconnectés. Si le spécialiste veut replacer *TraiterAchat* dans un autre "réseau" il doit donc au préalable avoir défini ce qu'était *TraiterAchat* : il doit comprendre toute la complexité de ce dernier, connaître ses "points d'entrée et de sortie"... tout est visible. Il n'y a pas d'élément qui englobe d'autres éléments et qui redistribue les messages qui lui parviennent à ces éléments englobés.
>
> 
>

Pour ces qualités vis-à-vis de la réutilisation, le paradigme objet

est aujourd'hui fortement présent dans les approches par méta-modèles.

2.2 La classification de Morch

Les mécanismes de flexibilité utilisés par les systèmes de workflow ne leur sont pas exclusifs. Dans son étude sur la malléabilité, Morch [MOR 94][MOR 97] a défini une classification des moyens pour rendre une application ou un système informatique plus malléable. Si l'emphase est portée sur l'implication des utilisateurs et donc sur les interfaces (ou interactions) homme-machine (IHM), l'étude reste toutefois dans la domaine de la flexibilité.

Trois approches sont des supports potentiels à la malléabilité : la conception participative, la réutilisation logicielle et la programmation par les utilisateurs.

- La conception participative : Elle a pour but de favoriser les interactions entre les utilisateurs et les informaticiens/concepteurs dans le processus de développement. Même si cette technique est intéressante et est utilisée pour la conception de certains WfMS, elle ne s'insère pas dans le contexte de la modification dynamique évoquée dans le chapitre 1.
- La réutilisation logicielle : Elle regroupe l'utilisation de bibliothèques de fonctions, l'héritage de classes et les générateurs d'applications. Toutes ces méthodes sont bien sûr présentes dans le monde du workflow. L'héritage des classes a déjà été évoqué avec l'apport du paradigme objet. La bibliothèque de fonctions peut être mise en correspondance avec les applications utilisées par les WfMS. Enfin les solutions d'EAI, qui ne sont pas (encore tout à fait) des WfMS mais qui emploient la modélisation workflow, peuvent être considérées comme des générateurs (de morceaux) d'applications.
- La programmation par les utilisateurs : Elle consiste à offrir aux utilisateurs un langage pour qu'ils puissent spécifier de nouveaux besoins. C'est une technique très puissante. Les méta-modèles workflow et les éditeurs graphiques associés sont issus de cette technique. Dans cette optique, il faut que le méta-modèle reste suffisamment compréhensible par les utilisateurs... à ce méta-modèle de proposer des concepts facilement compréhensibles ou encore de présenter une malléabilité hiérarchique. Pour cette dernière approche on retrouve des techniques comme l'encapsulation du paradigme objet ou encore le paramétrage (approche ponctuelle).

Les approches utilisées par les concepteurs de WfMS pour y apporter de la flexibilité ne sont pas spécifiques au workflow et se retrouvent donc dans d'autres domaines.

2.3 Notre choix : l'approche par méta-modèles

D'autres approches peuvent apporter de la flexibilité comme les systèmes multi agents [TAR 97] ou l'approche par prototype [MAN 93]. Ce sont néanmoins des approches moins répandues surtout dans le domaine du workflow ou des groupware. Une solution de coopération/interopérabilité basée sur ces approches nous semble pour l'instant peu utile.

Les techniques utilisées par les concepteurs de WfMS pour rendre ceux-ci flexibles ne sont pas propres au domaine du workflow. Nous considérons l'approche par méta-modèle comme la plus puissante. En intégrant les caractéristiques du paradigme objet, elle regroupe la réutilisation logicielle et la programmation par les utilisateurs.

On trouve différentes utilisation des méta-modèles dans le domaine plus général du génie logiciel : la description d'une application à l'aide d'un méta-modèle spécifique au domaine de l'application pour la génération de l'implémentation [LES 98], l'échange de spécifications entre ateliers de génie logiciel (AGL) [BOU 00b], la description d'interfaces utilisateur à travers un méta-modèle dédié [PAR 01]... L'usage des méta-modèles tel qu'il en est fait dans l'approche par méta-modèles reste toutefois plus fréquent dans le domaine du workflow. Cela est bien sûr dû à la nature particulière des WfMS dont l'objectif est de gérer tout type d'activités. Les méta-modèles servent à concevoir *mais aussi à modifier* des activités en cours. Dans les autres cas d'utilisation de méta-modèles, on demeure

dans un contexte de pure conception, donc statique. L'utilisation dynamique qui est fait dans les WfMS semble donc être est très indiquée pour le support aux activités d'une organisation.

L'évolution vers des systèmes informatiques très spécifiques aux domaines des entreprises ou des organisations (enseignement à distance, télécommunication, médical, ...) évoquée précédemment n'est donc pas incompatible avec l'approche par méta-modèles qui peut s'appliquer à tout type de systèmes. **Notre choix d'approche sur laquelle se reposera notre solution d'interopérabilité est donc une structuration en trois niveaux associée à des mécanismes réflexifs (changements dynamiques) accessibles aux utilisateurs (éditeurs).** Les spécificités des WfMS (simulation, optimisaton, service transactionnel, politique vis-à-vis des modifications dynamiques, ...) pourront être prises en comptes dans des travaux futurs mais pas dans ceux présentés dans ce travail. Nous réaffirmons encore une fois notre position : notre objectif est d'abord de proposer une solution pour l'architecture commune à une grande partie des futurs systèmes informatiques qui seront les supports aux organisations. Par la suite, nous utilisons le terme *méta-groupware* pour faire référence à ces systèmes.

Avant de présenter notre solution, nous allons montrer qu'aucune des solutions d'interopérabilité présentées dans le chapitre 2 n'est adaptée à la structuration précédente. Cette démonstration a pour principal intérêt de mettre en avant les faiblesses de ces solutions (vis-à-vis de notre contexte) mais aussi certaines qualités que nous ré-utilisons dans notre solution. Pour ce faire et pour présenter très clairement notre problématique, un scénario exemple est présenté dans la partie 4. Il sert, dans les chapitres 4, 5 et 6 de support d'illustration de notre proposition. Pour faciliter la compréhension de cet exemple, la partie 3 décrit précisément le concept de méta-modèle et l'intégration de modèles "actifs" dans un système.

3 Méta-modèle

Cette partie commence par un retour sur la définition de méta-modèle. La définition est générale et englobe tout type de méta-modèles. Nous réduisons ensuite cette définition avec la description des caractéristiques communes à tous les méta-modèles. La présence de différents niveaux dans le contexte de méta-modélisation rend parfois les propos difficiles. Pour simplifier ces derniers, nous utiliserons les termes M3, M2, M1 et M0 de l'architecture à quatre niveaux de l'OMG, objet de la section suivante. La dernière section traite de l'opérationnalisation de méta-modèles. Nous définissons la structure générale des éléments informatiques qui entrent en jeu pour rendre opérationnel un méta-modèle et surtout les liens entre ces éléments. C'est un aspect primordial pour nos travaux car l'opérationnalisation d'un méta-modèle est à la base de la construction d'un WfMS ayant adopté une approche par méta-modèle (meta-groupware). C'est ce qui nous permettra dans les chapitres suivants de construire une solution qui bénéficie de la structuration en trois niveaux.

3.1 Définition

Le terme méta signifie *derrière*, *qui vient après*, ou *au dessus*. Le mot où il apparaît pour la première fois est méta-physique, cité par Platon dans le mythe de la caverne (République – Livre VII). Le terme méta-modèle est un terme issu de l'informatique. Il n'a pas encore de définition dans un dictionnaire tel que le Larousse. Selon la signification de méta, méta-modèle correspondrait à *au dessus ou derrière le modèle*. Ceci reste très flou. La définition commune qui est attribuée au terme *méta-modèle*, et qui est plus précise, est "un modèle de modèle". [BOU 00] [LES 98] [GEI 97] [BLA 00]. La compréhension de cette définition nécessite la connaissance de celle de modèle. Un *modèle* est une *structure formalisée* utilisée pour rendre compte d'un ensemble de phénomènes qui possèdent entre eux certaines relations (*le petit Larousse 1999*). Le terme "formalisme de modélisation", souvent employé pour définir UML, prend alors toute son ampleur : il permet de formaliser une structure correspondante à des phénomènes et leurs relations. Néanmoins formalisme et formaliser présentent par définition des différences :

- Formalisme indique un attachement fort aux formes,

- Formaliser c'est poser explicitement dans une théorie déductive les règles de formation des expressions ou formules, ainsi que les règles d'inférence suivant lesquelles on raisonne (*le petit Larousse 1999*).

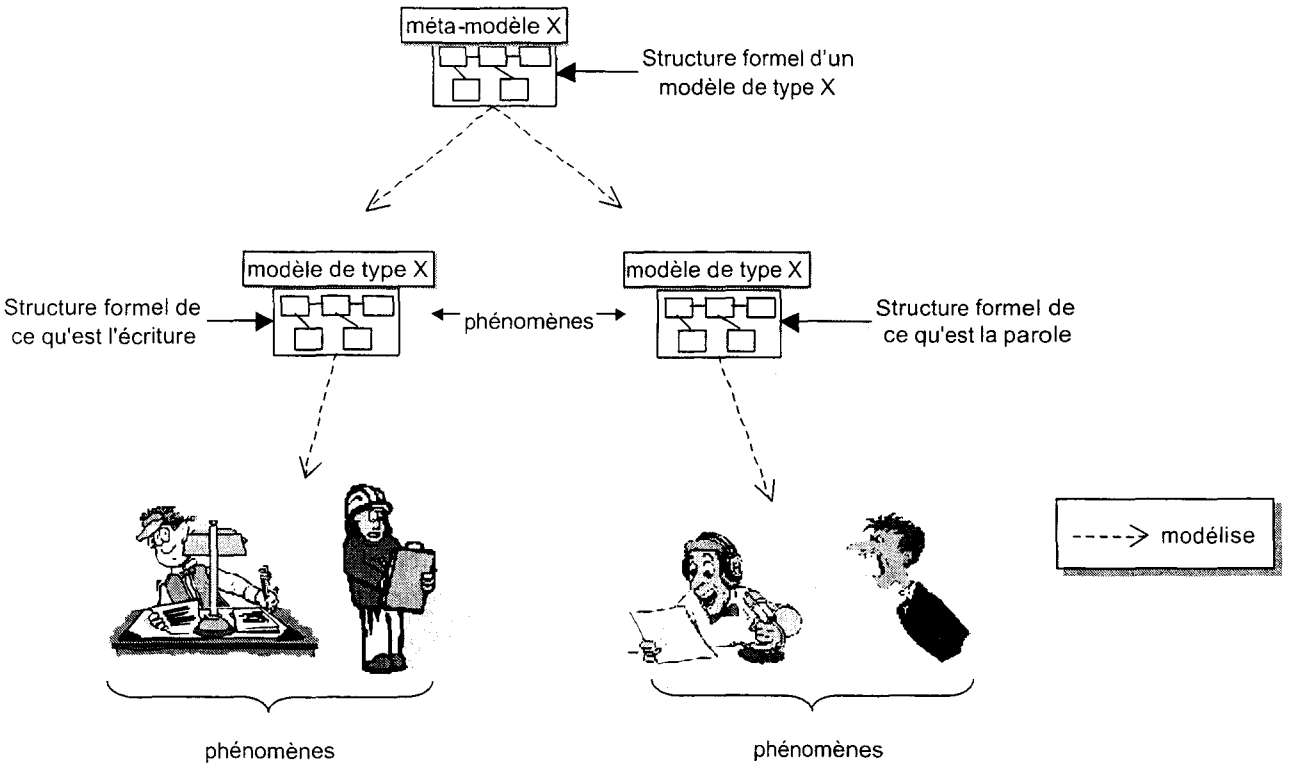


figure 9. Phénomènes, modèles et méta-modèles

Formaliser c'est s'attacher au contenu alors que le formalisme s'attache aux formes. On en déduit qu'un formalisme de modélisation est employé en vue de décrire le "contenu" de phénomènes en s'attachant aux formes. L'évocation du terme "formalisme de modélisation" n'est pas fortuite, elle est souvent rattachée aux méta-modèles. Si on reprend les définitions de méta-modèle et de modèle, un méta-modèle est une structure formalisée utilisée pour rendre compte de phénomènes qui sont une structure formalisée utilisé pour rendre compte d'un ensemble de phénomènes qui possèdent entre eux certaines relations. Il s'agit d'une structure formalisée qui permet de représenter des modèles (cf. figure 9).

3.2 Caractéristiques communes aux méta-modèles

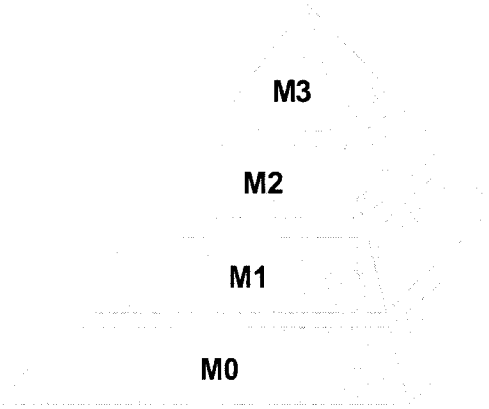
"Un méta-modèle est un formalisme de modélisation" paraît donc une conclusion logique. Cette affirmation n'est qu'à moitié vraie. Un méta-modèle est plus précisément la formalisation du formalisme de modélisation, la description précise des éléments explicites et implicites qui le constituent. Les programmes de simulation biologique, psychologique... sont appelés des modèles. Un langage de programmation permet de définir des modèles. La description formalisée d'un langage de programmation est donc un méta-modèle.

Pour les WfMS, un méta-modèle est la description précise de ce qui permet de définir des modèles de workflow. Pour certains WfMS, le méta-modèle n'est pas nécessairement un formalisme de modélisation, dans le sens où on l'entend habituellement, c'est-à-dire graphique. Il ne s'agit pas non plus à proprement parler d'un langage de programmation. Des systèmes comme WASA [MED 95] ont leurs modèles décrits dans des fichiers textes. La description respecte une grammaire particulière. La description de cette grammaire est-elle le méta-modèle ? Question qui se pose aussi pour les langages de programmation et leur grammaire associée. Pour les formalismes "graphiques" de modélisation, le méta-modèle correspond-il à la description des artefacts graphiques et des liens entre eux ?

A travers notre expérience de la méta-modélisation, de notre recherche bibliographique et des discussions entendues, **la seule chose commune à tous les méta-modèles est la description des concepts de modélisation.** Pour un WfMS, la description de la grammaire utilisée pour définir des modèles n'est pas la définition des concepts mais plutôt la description du moyen de les utiliser. Le méta-modèle UML ne définit pas "formellement" les artefacts graphiques qu'il utilise. Seuls les concepts de modélisation sont formalisés. Dans ce contexte, le méta-modèle d'un langage de programmation n'est pas obligé de faire apparaître la grammaire d'écriture de programmes.

3.3 L'architecture à quatre niveaux de l'OMG

Pour le reste du document, nous baserons notre analyse sur l'architecture pyramidale définie par l'OMG où elle classe les méta-méta-modèles, les méta-modèles, les modèles et les données utilisateurs. Le niveau M*x* permet de définir la structure et le comportement d'éléments du niveau M*x-1*.



- Les données utilisateurs, désignés par M0, correspondent à la description de phénomènes par un "système" informatique (voir l'encadré page 80).
- Un modèle, désigné par M1 (**M**odèle), correspond à la définition, ou permet la description de la structure et du comportement ("raisonnement") d'un ensemble de données utilisateurs.
- Un méta-modèle, désigné par M2 (**M**éta-**M**odèle), correspond à la définition, ou permet la description de la structure et du comportement d'un ensemble de modèles.
- Un méta-méta-modèle, désigné par M3 (**M**éta-**M**éta-**M**odèle), correspond à la définition, ou permet la description de la structure et du comportement d'un ensemble de méta-modèles.

Il n'y a que deux types d'opérations dans cette architecture :

La modélisation. Elle correspond à la description de phénomènes du monde "réel". Le "moteur" d'une modélisation est un méta-modèle dont les concepts, appelés aussi entités, permettent de modéliser une classe de phénomènes. On classifie les phénomènes par leur structure et leur comportement. Des phénomènes d'une même classe ont donc en commun la structure et le comportement définis par la classe. Plus précisément, chaque phénomène sera décrit par un élément qui sera une réalisation de la précédente classe. Cet élément est appelé *instance* dans le paradigme objet. Lors d'une modélisation, nous utilisons les entités d'un méta-modèle (M2) pour définir une classe (M1) qui permet de créer des éléments (M0) correspondant chacun à un phénomène. Ces éléments sont "similaires" (d'où la classification).

La méta-modélisation. Elle correspond uniquement à la description de classes de phénomènes. Les définitions précédentes de classes avaient pour but de décrire des phénomènes grâce aux réalisations des classes. Ici il s'agit de décrire des classes issues de n'importe quel méta-modèle. Pour cela, il nous faut classifier les classes de phénomènes, c'est-à-dire décrire les structures et les comportements communs des classes. Chaque description structurelle et comportementale correspond à un concept d'un méta-modèle. Pour définir ces méta-modèles, opération appelé méta-modélisation, on utilise un méta-méta-modèle (m^2-modèle) constitué de méta-entités. Lors d'une méta-modélisation, nous utilisons les méta-entités d'un m^2-modèle (M3) pour définir les concepts d'un méta-modèle (M2) qui permettent de décrire, à travers la réalisation de ces concepts (M1) tout type de phénomènes.

M3	exemples	**MOF**	**CDIF**	**Entité-Relation**	**sNets**
	éléments	Class, Package, Association	Universe,	Entité, Relation	Node, Link
M2	exemples	**Java**	**UML**	**méta-modèle du WfMC**	**DARE**
	éléments	Class, Attribute, Method, Package, Modifier, ...	Class, Association, Reference, Attribute, ...	Process, ProcessModel, Resource, ...	Task, Role, Tool, Action, ...
M1	exemples	**Une application Java**	**Un modèle UML**	**Un modèle de workflow**	**Un modèle de tâches**
	éléments	java.awt.Frame, myAppli.Serveur, ...	Caissier, Personne, ...	AchatMatériel, RédigerRapport, ...	Vote, Co-rédaction, Réunion, ...
M0	exemples	**Des objets Java**	**Des objets UML**	**Des cas de workflow**	**Des activités coopératives**
	éléments	f = new java.awt.frame ("Saisie ...");	C:Caissier , Alain: Personne	AchatMatériel { "DellXPS", 19/06/2002, ...}	Vote { "Loi12", Greg, Alain, ... }

tableau 1. Exemples pour chaque niveaux de l'architecture de modélisation de l'OM G

Le tableau 1 donne des exemples d'éléments de chaque niveau. Y sont présents les méta-méta-modèles MOF, CDIF, les sNets et Entité-Relation. Si le dernier est courant, ce n'est pas le cas des trois autres. Le *Meta-Object Facility* (MOF) [OMG 00], sujet du chapitre 5, est un m^2-modèle défini par l'OMG dans le but de standardiser l'accès aux modèles de données. *CASE Data Interchange Format* (CDIF) [ERN 97], créé par le comité de standardisation industriel EIA (Electronic Industries Alliance) [EIA 02] est un m^2-modèle défini pour faciliter les échanges entre outils de modélisation. Signalons que CDIF a été abandonné en décembre 1998. Les sNets [BEZ 94] sont issus des travaux de recherches de Bézivin sur la méta-modélisation et l'utilité de celle-ci dans la conception d'applications. Le méta-méta-modèle des sNets a pour particularité de se vouloir minimal (2 méta-entités et une méta-relation).

Sur le tableau 1, les flèches pour le niveau M3 indiquent que le MOF ou CDIF peuvent décrire tous les méta-modèles. Alors que le méta-modèle Java est fait pour définir des classes Java (et pas des classes UML). Tout langage peut servir pour définir des méta-modèles, mais il faut être conscient que le langage est utilisé dans un contexte de méta-modélisation. Le MOF, CDIF et les sNets sont étiquetés "méta-modélisation". Les utilisateurs en ont donc conscience.

Précision sur le terme "M-zéro":

Il y a deux précisions à apporter sur le niveau M-zéro : la terminologie et la double vue.

Terminologie. L'OMG a défini le niveau M-zéro pour la première fois dans les spécifications du MOF. Dans celles-ci, le niveau M-zéro désigne *les informations que l'on souhaite décrire*. Dans nos travaux, nous délimitons le niveau M-zéro aux informations informatisées, c'est-à-dire celles que l'on peut manipuler à partir d'un support informatique. Notre choix est plus proche de la définition du niveau M-zéro donnée dans les spécifications d'UML – *instances de modèles* [UML 01][WAR 02]. Il est très important de se rappeler de cette restriction pour le reste de l'ouvrage.

Double vue. La partie 3.4.3.2 montre qu'une information informatisée de type M-zéro a deux "types" : sa base générique et son type de définition. La base générique est généralement un élément du méta-modèle alors que le type de définition est souvent un élément de modèle (issu du précédent méta-modèle). L'information doublement typée peut alors être considérée comme une instance d'un méta-modèle (et donc annotée M1) ou comme une instance d'un modèle (alors annotée M-zéro). Toutefois seul le créateur du méta-modèle accède directement à la base générique et la manipule explicitement. A l'inverse, l'utilisateur (du méta-modèle) ne considère que le type de définition. Les annotations M1 et M-zéro correspondent alors chacune à un point de vue particulier :

- M1 → le point de vue du créateur (ou point de vue physique),
- M-zéro → le point de vue "utilisateur" (ou point de vue logique[28]).

Cette dualité de point de vue s'illustre très bien avec l'exemple UML du tableau 1 :

- [vue logique] L'objet C est vu comme une instance de *Caissier* : C est de niveau M-zéro.
- [vue physique] L'objet C est vu comme une instance de *UML::Object* : C est de niveau M1.

Dans cet ouvrage, le terme M-zéro fera implicitement référence au point de vue logique. Le chapitre 5 reviendra longuement sur la mise en œuvre (vue physique) du niveau M-zéro.

3.4 Opérationnalisation d'un méta-modèle

Nous utilisons le terme "opérationnalisation" pour désigner le fait qu'un méta-modèle devient opérationnel, c'est-à-dire prêt à entrer en activité, à réaliser des opérations. Quelle est l'activité d'un méta-modèle ? Créer des modèles. La description d'un méta-modèle sur papier ne rend pas opérationnel celui-ci. Par contre si la description est effectuée à travers un ensemble de classes Java, le méta-modèle est opérationnel : les classes Java étant instanciables, les instances des précédentes classes correspondent à des instances d'un méta-modèle, c'est-à-dire des modèles. Par exemple, le concept *ProcessType* est décrit à travers la classe Java `ProcessType`. Une instance de cette classe correspond à la définition d'un type de processus. Les outils MOF (cf. chapitre 5) permettent aussi l'opérationnalisation d'un méta-modèle. Celui-ci est décrit soit dans un diagramme de classes UML ou dans une déclaration textuelle MODL (Meta Object Definition Language [OMG 99]). Un outil MOF traduit ensuite cette description en un ensemble d'interfaces IDL et de classes Java qui constituent un méta-modèle opérationnel.

Nous avons expliqué que la description de l'opérationnalisation permettrait de tirer profit des systèmes adoptant une structuration en trois niveaux. Il y a trois aspects à évoquer dans cette opérationnalisation :

- L'accès informatique : comment sont créés les modèles, comment accéder au modèle...
- Le référentiel : le service de création, le point d'entrée pour l'accès aux modèles.
- Le niveau M-zéro : comment sont créées les "instances" de modèles. C'est l'aspect le plus complexe et le plus intéressant.

[28] Riehle [RIE 01] est à l'origine de l'utilisation des termes physique et logique dans ce contexte. Dans notre cas, nous employons toutefois le terme physique sans avoir obligatoirement à l'esprit une projection vers un support de réalisation.

3.4.1 L'accès informatique

Nous avons déjà évoqué l'accès informatique dans la section 2.1.2 au sujet de l'implémentation des trois niveaux. Il s'agit de la façon dont on accède aux entités d'un méta-modèle opérationnel et à ses instances à partir d'un programme. Par exemple, si l'opérationnalisation a donné lieu à la création de classes Java, la création d'un modèle consiste à instancier ces classes et à affecter ensuite des valeurs aux attributs (des instances). L'accès informatique est ici un ensemble de classes Java (correspondant aux entités du méta-modèle).

L'accès informatique est problématique dans un contexte d'interopérabilité. Lorsqu'un système a besoin d'accéder aux modèles "actifs" d'un autre système, l'implémentation de cet accès nécessite de connaître les entités du méta-modèle et l'accès informatique. Quand la définition des entités est faite à travers des classes Java, l'accès informatique se déduit directement : ce sont ces mêmes classes Java. Dans le cas où la définition des entités est réalisée à partir du formalisme UML et l'opérationnalisation consiste en un ensemble de classes C++, créées à partir d'une projection UML→C++ effectuée "à la main", l'accès informatique est plus problématique. En effet, pour les concepteurs d'un système "externe", l'implémentation de l'accès à ce méta-modèle opérationnel nécessite de connaître les concepts du méta-modèle et les règles de projection. Ce problème est réel. Nous verrons d'ailleurs dans le chapitre 5 que l'objectif du MOF est de supprimer le problème des règles de projection.

3.4.2 Le référentiel

Dans la suite de ce manuscrit, nous serons souvent amenés à parler de "référentiel". Le terme "référentiel" est un raccourci pour l'expression référentiel de modèles, traduction de l'anglais "model repository". Bien que l'expression "dépôt de modèles" eut été une meilleure traduction, le terme référentiel évoque en plus une référence, et donc dans le contexte d'un système distribué, un endroit où se trouvent les modèles.

Le référentiel est la porte d'accès à tous les modèles que peut contenir un système. C'est le composant ou le service auquel on s'adresse pour connaître les modèles existants mais aussi pour en créer de nouveau. La définition n'intervient pas dans celle du méta-modèle, mais dans les règles de projection en rapport avec l'opérationnalisation. Sur l'exemple de l'opérationnalisation en Java du méta-modèle général du workflow (cf. 1.3), la classe `Repository` est une possibilité de définition du référentiel. Elle contiendrait, par exemple, les opérations `create_ProcessType`, `allProcessType`, `create_ResourceType`, `allResourceType`... L'implémentation de l'opération `create_ProcessType` consiste ici à instancier la classe `ProcessType`.

3.4.3 Le niveau M-zéro

L'accès informatique et le référentiel sont essentiels dans l'exploitation des structures à trois niveaux. Ils demeurent toutefois un problème assez récurrent du passage de l'aspect conceptuel à l'implémentation. La construction du niveau M-zéro est quant à elle plus originale. Elle consiste à une définition explicite de mécanismes d'instanciation. À l'aide d'un exemple basé sur le méta-modèle simplifié du WfMC, nous donnons un aperçu de mécanismes d'instanciation nécessaires pour la construction du niveau M-zéro. En étudiant l'élaboration de tels mécanismes dans les WfMS flexibles mais en s'inspirant aussi de la méta-programmation et de certains aspects de la modélisation UML, nous définissons notre vue du niveau M-zéro.

3.4.3.1 Exemple

Les objectifs de l'opérationnalisation d'un méta-modèle peuvent être multiples : fournir une vue particulière de la structure d'un ensemble d'informations déjà existantes, stocker des modèles de conception, définir des types d'éléments que l'on souhaite manipuler dans un système... Le dernier exemple est celui des WfMS flexibles ayant adopté une approche par méta-modèle. L'éditeur de modèles du WfMS donne aux utilisateurs la possibilité de créer des modèles (où apparaissent des types de processus, de ressources, d'activités...) qui permettent de gérer des flux de travail et les flux

d'informations dont les types sont décrits dans les précédents modèles. Par exemple, l'utilisateur crée le modèle où apparaît le type de processus *TraiterAchat*, pour gérer par la suite des processus tels que *AchatDellXPS*. L'élément informatique qui gère *AchatDellXPS* dans le système est (ou peut être vu comme) une instance de *TraiterAchat*. En résumé, *AchatDellXPS* est une instance de *TraiterAchat* qui est une instance de *ProcessType* (entité du méta-modèle). C'est cette double instanciation du méta-modèle qu'implique le niveau M-zéro.

À notre connaissance il n'y a pas de langage ni de formalisme qui permette de décrire un méta-modèle de manière à indiquer les règles de double instanciation. Les mécanismes de gestion du niveau M-zéro sont en général à la charge du concepteur, alors que l'instanciation du méta-modèle est gérée par le support d'implémentation choisi. Cette affirmation peut s'illustrer sur un exemple.

Si le concepteur décrit *notre* méta-modèle de workflow (c'est-à-dire la modélisation UML des concepts définis par le WfMC) à l'aide de classes Java, les entités telles que *ProcessType* ou *ResourceType* sont instanciables grâce au mécanisme d'instanciation du langage Java. Les instances des classes Java `ProcessType`, `ResourceType`, etc. forment des modèles. Par contre, le langage Java ne propose pas de mécanismes d'instanciation pour ces modèles : les instances de classes ne sont pas instanciables. La situation est identique dans le cas d'une projection en Java d'un méta-modèle décrit en UML. Toutefois, il est possible de créer ses propres mécanismes d'instanciation (cf. figure 10) : pour chaque entité/classe Java est créée une classe Java additionnelle qui fournit une structure et un comportement générique pour les instances des instances de l'entité. Cette construction s'inspire du patron de conception *type-objet*. Par exemple, est associée à la classe `ProcessType`, la classe `Process` dont les instances (ex : *AchatDellXPS*) seront les instances de `ProcessType` (ex : TraiterAchat). Les objets de type `Process` ne sont des instances Java de `ProcessType`. Par contre, elles le sont de notre point de vue (c'est-à-dire dans notre système). La classe `Process` définit un attribut `type` dont le type de valeur est `ProcessType`. Cet attribut correspond au type de processus. La classe `ProcessType` définit une méthode `newInstance` dont l'invocation sur `TraiterAchat` crée une instance de `Process` et affecte l'objet `TraiterAchat` à l'attribut type de cette nouvelle instance.

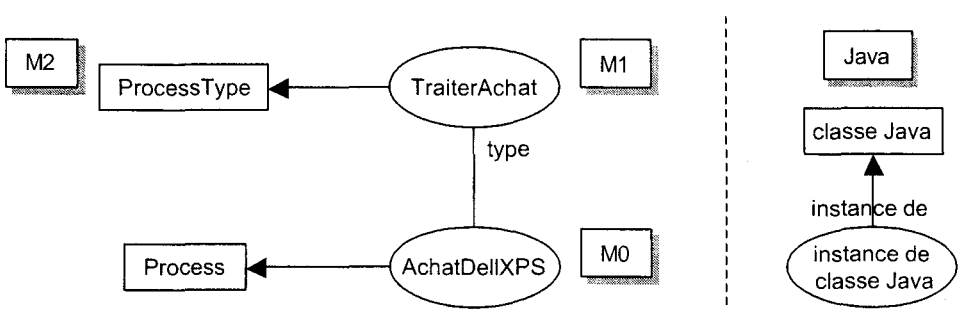


figure 10. Simulation en Java de la double instanciation

En résumé, la classe `Process` fournit une structure et un comportement générique qui sont ensuite caractérisés, spécialisés par un type, c'est-à-dire une instance de `ProcessType`. La classe `Process` fournit une partie de l'implémentation du niveau M-zéro. Il reste à définir les classes `Activity`, `Resource`, `Participant`, ...

3.4.3.2 La séparation entre base générique et type

D. Manolescu [MAN 98] présente différents patrons de conception pour construire des systèmes de workflow adaptatifs (modèle dynamique, gestion des exceptions, ...). C'est une restriction des différents patrons de conception utilisables pour construire des "systèmes objets" [GOE 00]. D. Manolescu préconise le patron de conception *type-objet* (précédemment cité) pour implémenter le support de modèles dynamiques, c'est-à-dire la structuration en trois niveaux. Ce patron a pour objectif *d'étendre un système sans programmer de nouveaux types* [JOH 98] :

> Dans un langage de programmation objet, une instance est fortement attachée à son type, c'est-à-dire sa classe. Si les modèles sont traduits en classes, la modification dynamique d'un modèle se traduit en modification dynamique de classes. C'est une capacité que n'ont pas tous les langages objets (en particulier les plus utilisés commercialement). *Type-objet* (ou type-instance) permet de découpler l'instance de sa classe. Son type devient, grâce à ce patron de conception, une instance d'une classe. On est donc en présence de deux classes : la classe *<Type>* et la classe *<Instance>*. Une instance d'*<Instance>* a pour type une instance de *<Type>*. Pour les éléments M-zéro, il y a la structure primaire gérée par la classe *Instance* et le type de définition géré par la classe *<Type>*.

Le *pattern type-objet* est utilisé dans les systèmes Endeavor (deux classes Java *MetaClass* et *ObjectStore*) [Teamware], InConcert [ABO 94], COW, DARE (ces deux systèmes sont détaillés plus loin) et sera utilisé dans Zope3Flow [ZOP 02]. Les travaux de E. Breton [BRE 02] sur la définition de liens entre le standard de modèles du WfMC[29] (M1) et le standard de moteur d'exécution du WMF[30] (M-zéro) font clairement apparaître ce *pattern* (cf. figure 11). *Workflow Process Definitio* n permet de définir les types pour les instances de *WfProcess*. Ces deux "éléments" sont pourtant représentés comme des classes MOF[31].

Cette séparation entre la base générique et le type dans le *pattern type-object* apparaissait déjà auparavant dans d'autres travaux concernant la méta-programmation et les systèmes réflexifs. J. Ferber [FER 89] indiquait que les langages à base de méta-classes proposaient une réflexion informatique moins puissante que les langages à base de méta-objet. En effet, dans l'approche par méta-classes, les messages que reçoivent les instances sont toujours gérés par le même interpréteur. Alors que dans l'approche par méta-objet, les messages d'une instance sont gérés par son méta-objet que l'on peut changer. Cette approche *réifie* une partie supplémentaire du comportement des instances (l'interprétation des messages reçus). L'utilisation de méta-objets permet d'aller plus loin dans la définition de la structure et du comportement générique d'un groupe d'instances sans pour autant se référer aux types.

La dualité type/base générique apparaît (logiquement) dans le domaine de la (méta-) modélisation. Dans les spécifications d'UML [UML 01], il est dit que *la plupart des concepts de modélisation ont un caractère double et que celui-ci est modélisé par deux éléments : un qui représente le descripteur générique et l'autre les items individuels qu'il décrit.* Par ailleurs, G. Genilloud estime que les définitions d'UML concernant les termes *instance*, *classe* et *type* sont floues [GEN 00]. D'abord il indique qu'il ne s'agit pas de *la plupart* mais de *tous les* concepts de modélisation. Ceux qui n'ont pas ce caractère double sont victimes d'une omission (qu'il tente de réparer en partie dans [GENI 00] [WEG 00]). Ensuite, il reprend les notions de *type*, d'*instance* et de *template* d'ODP pour préciser les précédents termes (*instance*, *classe* et *type*). Un *type de <X>* est *un prédicat qui permet de caractériser un ensemble de <X>*. Un <X> qui est une *instance* d'un type est *un <X> qui satisfait le type*. Les <X> sont créés à partir du *<X> template* (qui correspond à la base générique) c'est-à-dire *la spécification des caractéristiques communes d'une collection de <X> avec suffisamment de détails pour qu'un <X> puisse être instancié en l'utilisant. <X> peut être 'n'importe quoi' qui a un type.* Encore une fois, on trouve la création d'une base générique (avec les *templates*) et le fait de pouvoir classifier les éléments grâce aux *types*. La création de "types" en modélisation UML se fait à l'aide de concepts de type et la création d'instances à l'aide de concept d'instance.

[29] Workflow Management Coalition

[30] Workflow Management Facility

[31] Ces travaux sont importants car, même s'ils diffèrent des nôtres (notre "postulat" de départ est qu'il est impossible d'avoir une norme ou un standard), ils confirment notre volonté d'expliciter les liens entre les modèles (M1) et leurs instances (ou réalisations) (M-zéro).

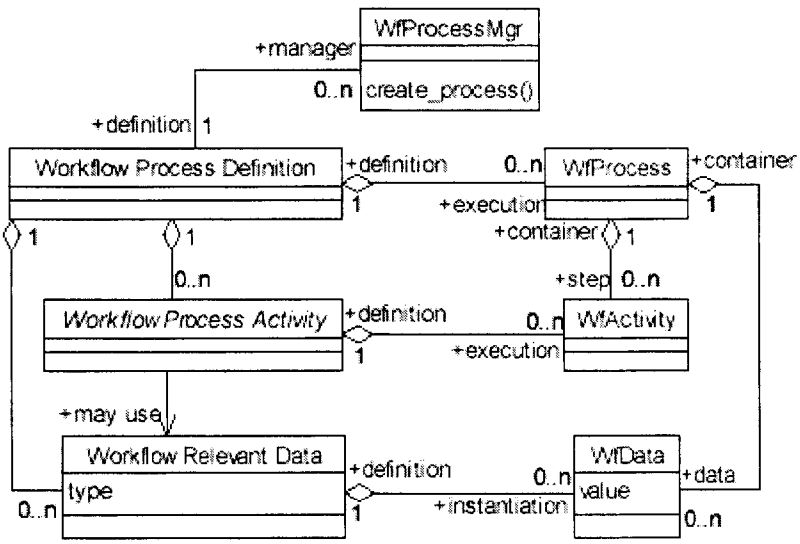


figure 11. Modélisation des WfMS "standards"

3.4.3.3 La définition conceptuelle du niveau M-zéro

Un concepteur qui souhaite définir précisément un méta-modèle, c'est-à-dire avec son comportement au niveau M-zéro, se retrouve dans une situation difficile. Soit il transfert au sein du méta-modèle ses préoccupations d'implémentation : les entités sont définies pour créer soit des modèles soit des instances de ces modèles (comme pour les travaux de E. Breton [BRE 01]). Soit il définit un méta-modèle (seulement les entités destinées à créer des modèles) ET un modèle de "fonctionnement", qui reprend en partie le méta-modèle (DARE [BOU 99]) : le modèle de fonctionnement est plutôt une représentation du patron de conception utilisé. Elle n'est pas une réelle description conceptuelle (des entités d'instance) et est, à notre avis, à éviter. Dans la première configuration, les liens très forts entre les entités de types et les entités d'instances n'apparaissent malheureusement pas en tant que tel : ce ne sont que des liens (et non pas des liens de 'typage sur les instances').

Les formalismes de méta-modélisation utilisés actuellement ne proposent pas de méta-entités spécifiques pour définir des entités de types ET des entités d'instances, et implicitement ne fournissent pas de méta-relations pour associer de telles entités. L'opérationnalisation n'est donc pas, d'un point de vue conceptuelle, exprimable. Ceci est probablement une erreur. De notre point de vue, une entité a conceptuellement deux aspects : un aspect pour ses instances et un aspect pour les instances de ses instances. Le deuxième aspect est la continuation du premier. Autrement dit, des entités permettent de décrire dans un premier temps *des structures formalisées d'ensembles de phénomènes* et dans un second temps des *phénomènes particuliers* de ces ensembles.

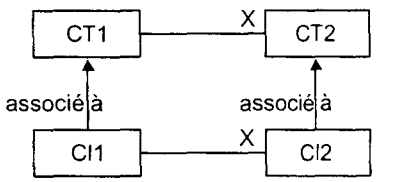
Pour l'instant nous allons définir les relations habituelles entre les entités destinées aux modèles et celles destinées aux instances de modèles. Nous reprenons les notions de concepts de type et d'instance d'UML que nous replaçons dans un contexte plus général (de méta-modélisation). Cette base, fondamentale, nous permettra dans les chapitres suivants de bénéficier de la structuration en trois niveaux lors de la conception d'une solution de coopération de méta-groupware. Nous les intégrerons à cet effet au méta-méta-modèle MOF (cf. chapitre 5).

3.4.3.4 Notre vision du niveau M-zéro

Nous appelons ***concepts d'instance*** (CI) les entités qui permettent l'instanciation des modèles (comme `Process`) à l'opposé du ***concepts de type*** (CT), équivalents d'*entités* (comme `ProcessType`) support à la création des modèles. Pour les éléments du niveau M-zéro (comme

AchatDellXPS), nous appelons ***type réel*** leur structure générique et ***type de définition*** le type de caractérisation. Pour l'instance de `Process` *AchatDellXPS*, `Process` est son type réel et *TraiterAchat* (l'instance de `ProcessType` référencée par l'attribut type) est son type de définition. Un concept d'instance est obligatoirement attachée à un concept de type[32] (il en est sa continuité). L'exemple le plus célèbre est celui du CT *Class* et du CI *Object*. Pour un méta-modèle, l'opérationnalisation implique la définition des CIs, en plus de celle des CTs.

Un CI reprend généralement une partie des liens du CT auquel il est associé. Les types de valeurs de ces liens répercutés sont les CIs associés aux CTs, types de valeurs des liens de départ. Par exemple, le concept de type (CT) Classe est associé avec le CT Attribut : une classe contient un ensemble d'attributs. Le concept d'instance (CI) Objet est associé au CI AttributInstance : un objet contient une instance de chaque attribut (une propriété) défini dans sa classe. Sur notre précédente projection Java, la classe `ProcessType` a un attribut `isComposedBy` dont le type est `ActivityType`. Cet attribut est répercuté sur `Process`, qui dispose d'un `isComposedBy` mais dont le type est `Activity`.



Pour les concepts d'instances, nous reviendrons sur les mécanismes à implémenter tels que la vérification de typage et l'héritage dans les chapitres 5 et 6. Pour l'instant, seules trois informations nous sont utiles :

1. La structure et le comportement des instances d'instances d'un concept de type (le seul type de concept généralement présent dans un méta-modèle) sont définis dans un concept d'instance.
2. La structure et le comportement précédents sont génériques. Pour une instance d'un CI, ils sont caractérisés par une instance du CT associé. Un concept d'instance définit toujours un lien de "type de définition" vers son CT associé.
3. Les liens définis sur un CT sont répercutés en partie sur son CI. Les noms peuvent bien évidemment changer.

La description précise du concept de méta-modèle et des implications de l'opérationnalisation nous permettent de mieux appréhender les enjeux de la coopération de groupware flexibles. La liste des mécanismes de coopération à fournir est maintenant dressée à l'aide d'un scénario exemple.

4 Scénario exemple

Dans cette partie est décrit un exemple de coopération entre le système DARE et le système COW. Ces deux systèmes ont été développés dans notre laboratoire. Ils ne sont pas pour l'instant le support d'une réelle étude de cas car ils sont à l'état de prototype. Ce scénario de coopération fournit un support pour mettre en exergue, dans la partie 5, les faiblesses des solutions d'interopérabilité sur la coopération de groupware flexibles et surtout permet de bien étudier les mécanismes nécessaires pour la coopération dans un contexte dynamique. Le scénario a pour dernier avantage de proposer deux méta-modèles fort différents[33].

---

[32] Un CI peut être rattaché à plusieurs CTs : la caractérisation nécessite d'affecter plusieurs types secondaires. Ex : Endeavor (2 CTs) [KAM 00].

[33] En effet DARE et COW, s'ils servent tous les deux à générer des environnements pour le travail coopératif (ce sont tous les deux des méta-collecticiels) appartiennent à des écoles de pensée différentes en matière d'approche du travail coopératif. DARE est le fruit d'une approche très sociable et cognitive du travail coopératif mettant en exergue le caractère fragile, négocié, temporaire, revisable de la coopération [SCH 00], alors que COW s'inscrit beaucoup plus dans une école de pensée de type organisationnel avec la division du travail et sa coordination. Cela sera explicité lors de l'examen des méta-modèles sous-jacents à DARE et COW.

Les systèmes DARE et COW et leur méta-modèle respectif sont dans un premier temps décrits. Est ensuite présenté le contexte où deux sociétés d'enseignement à distance doivent coopérer. Le contexte de ce scénario s'inscrit dans l'orientation de nos travaux. Pourtant il nous préciser un point important. Un des objectifs des travaux de notre laboratoire est de rendre COW coopératif à l'aide d'une fusion avec DARE (ce qui a d'ailleurs été l'objet d'un travail présenté dans la conférence EDOC [LEP 01]). Néanmoins il n'en est pas question ici. DARE est employé ici comme un WfMS. L'utilisation sort du champ applicatif de DARE. Mais il y a trois raisons qui permettent pourtant de se baser sur un tel exemple fictif :

- Nous maîtrisons parfaitement les méta-modèles de DARE et COW et leur opérationnalisation.
- L'utilisation de DARE n'est pas incohérente.
- Les différences entre les deux méta-modèles (issues des origines de conception) font de cet exemple un bon candidat pour notre étude.

4.1 DARE

4.1.1 Présentation

DARE (*Distributed Activities in a Reflexive Environment*) est un méta-collecticiel réflexif. Il permet à plusieurs utilisateurs, situés dans des endroits différents, de participer à une même activité. La participation à une activité se fait via un navigateur Internet.

DARE se focalise plus sur la coopération que sur la coordination (qui lui est l'intérêt principal des WfMS). Pour construire un cadre conceptuel adapté à la coopération, G. Bourguin, concepteur de DARE, s'est inspiré des théories sociales et plus particulièrement de la Théorie de l'Activité (cf. chapitre 1). Il en déduit qu'une plus grande liberté doit être laissée aux utilisateurs : la coordination des activités est au départ pré-déterminée mais surtout elle peut être à tout moment modifiable par le groupe d'utilisateurs (ce type de modifications doit d'ailleurs se faire lui aussi de manière coopérative). C'est une démarche opportuniste : selon le contexte, les utilisateurs décident de la suite. Cette approche est à l'inverse des WfMS où les actions sont pré-déterminés et seuls les "spécialistes" (responsables du management) peuvent modifier les règles de coordination, c'est-à-dire l'enchaînement des processus. De l'AT, G. Bourguin en retire aussi que les relations entre un participant et la communauté (les outils, le groupe, ...) doivent être médiatisées par des règles.

Toute la structure de DARE essaie d'être une synthèse de cette base théorique. Pour répondre à la nécessité d'évolution de la structure d'une activité, DARE intègre un niveau méta. À l'aide d'une interface utilisateur associée, ce niveau permet aux utilisateurs de modifier la structure d'une activité à l'exécution de celle-ci. Il est à noter que comme l'accès à ce niveau n'est pas réservé qu'aux spécialistes, la méta-activité apparaît comme toute autre activité, ceci afin de ne pas perturber les utilisateurs ET de rester une activité coopérative. Pour la coordination des activités, elle est en grande partie gérée par les utilisateurs. De manière similaire au workflow, une activité est la réalisation de plusieurs sous-activités. Là où les WfMS indique quelle est la sous-activité à effectuer, DARE laisse les utilisateurs choisir. Néanmoins, la liste de sous-activités a, au préalable, été définie dans le modèle de l'activité. Les utilisateurs ont donc plus de liberté[34]. Les relations entre participants et la communauté sont gérées dans DARE à travers des rôles et des droits d'accès aux outils.

[34] Dans une prochaine version de DARE, la possibilité de définir des règles de coordination sera présente. Si elle n'est pas présente dans le première version de DARE, c'est parce qu'elle ne constituait pas une priorité. De plus, la conception d'une telle fonctionnalité n'est pas facile car elle doit intégrée une certaine rigidité tout en intégrant aussi une certaine liberté.

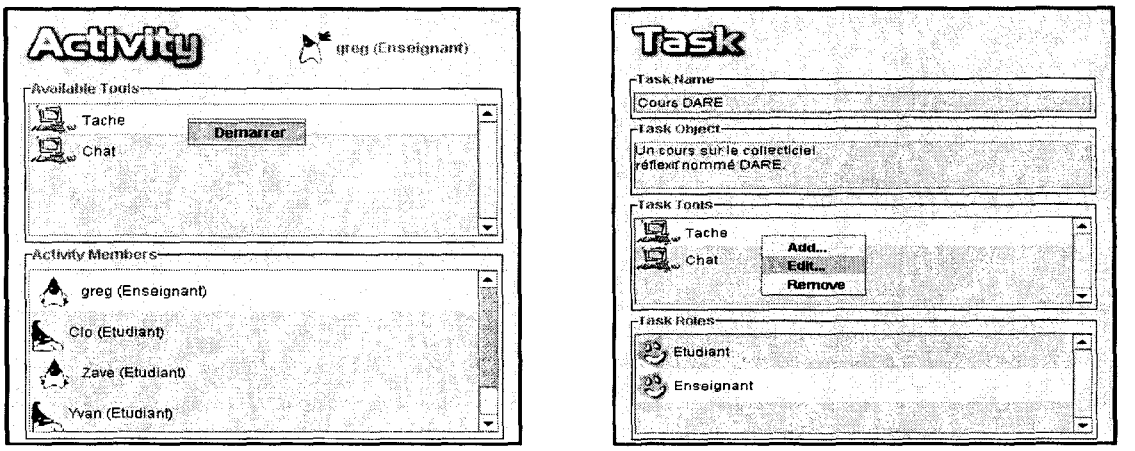


figure 12. Exécution du système DARE

4.1.2 Le méta-modèle de DARE

Les principaux concepts de DARE sont la Tâche, le Rôle et l'Outil. La figure 13 montre le méta-modèle complet de DARE. Ils sont exprimés dans un UML simplifié qui est la base du MOF (que nous verrons dans le chapitre suivant).

Concepts de type. Dans DARE, une activité à laquelle participent des utilisateurs est décrite par une tâche (*Task*). Une tâche indique les outils (*Tool*) et rôles (*Role*) disponibles. Une tâche peut aussi contenir d'autres tâches (par le biais de la référence *uses*). Un outil est techniquement réalisé par une applet Java, à laquelle est associée une liste d'opérations (*Operation*). Chacun opération est un morceau de code Smalltalk invoquant ou non des actions de l'outil. Une action (*Action*) est l'encapsulation d'une méthode de l'applet. Les droits d'accès aux outils sont déterminés à partir des rôles. L'élément principal de la définition d'un rôle est une liste de micro-rôles (*MicroRole*). Chacun d'eux est associé à un seul outil pour lequel il indique les opérations accessibles.

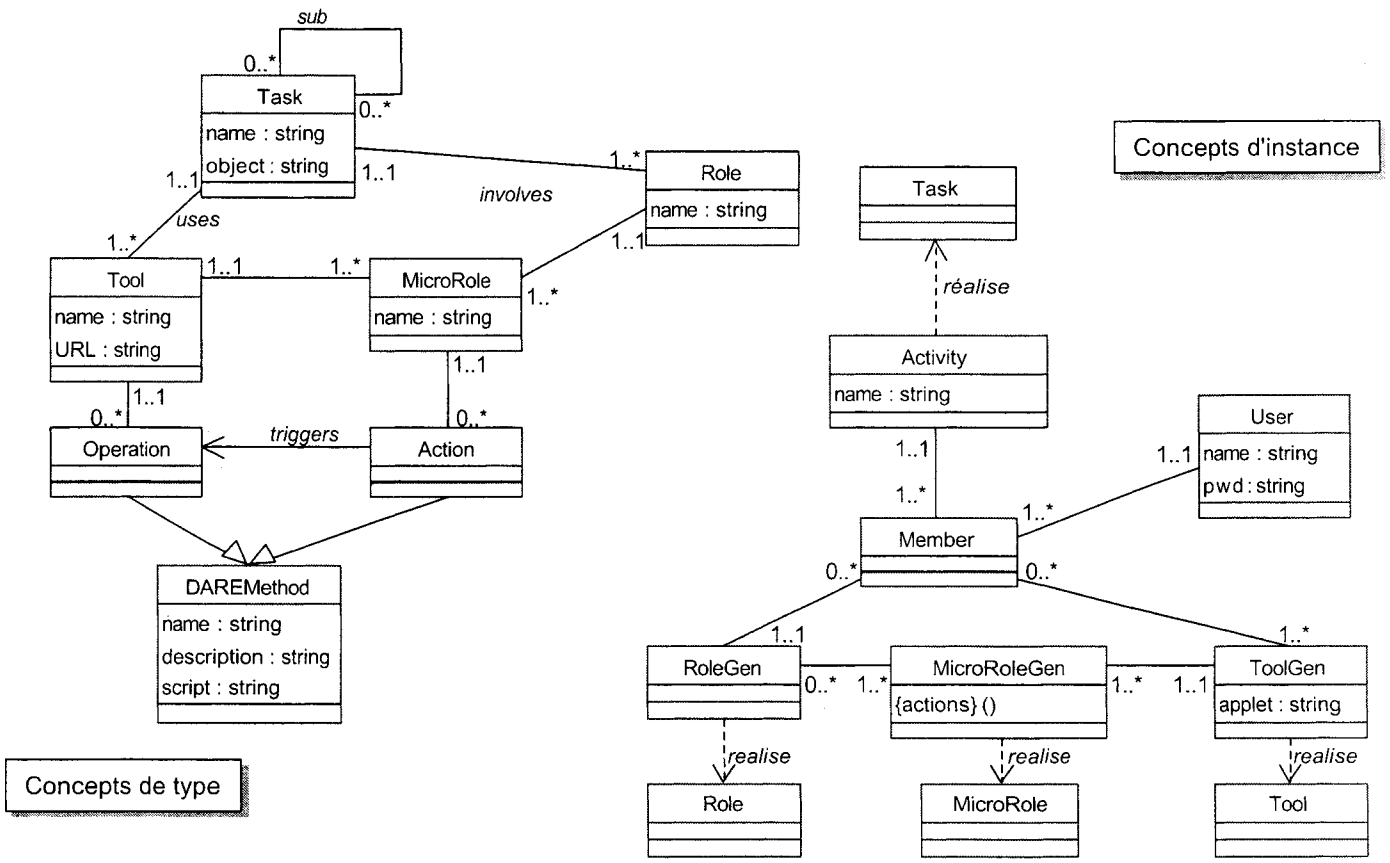


figure 13. Méta-modèle de DARE

Concepts d'instance. *Task*, *Role*, *MicroRole* et *Tool* sont respectivement réalisés par les CIs *Activity*, *RoleGen*, *MicroRoleGen* et *ToolGen*. Les deux concepts *Member* et *User* interviennent au niveau M-zéro mais ne sont pourtant pas des CIs. Chaque personne qui se logue sur le système DARE devient un utilisateur (*User*). Il crée une activité (à partir d'une Tâche) ou s'insère dans une activité déjà existante. Il devient alors un membre (*Member*) de celle-ci. Il appartient donc à une activité (*Activity*) pour lequel il remplit un rôle (*RoleGen*) et où il utilise des outils (*ToolGen* : une éxécution de l'applet spécifiée dans un outil). L'accès aux opérations de l'outil est géré par les micro-rôles (*MicroRoleGen*).

Les mécanismes réflexifs intégrés dans DARE sont clairement identifiés comme tels : ils sont basés sur le Meta Object Protocol [KIC 93] et sont construits grâce aux propriétés du langage Smalltalk 80 [GOD 89].

4.2 COW

4.2.1 Présentation

COW (Cooperative Open Workflow) [VAN 02] [VAN 02b] est un système de workflow coopératif. Son méta-modèle est très proche de celui du WfMC. Les différences conceptuelles qui apparaissent sont dues à l'ajout d'un aspect coopératif. Pour l'instant ces éléments additionnels ne sont pas opérationnels. Les fondements théoriques de COW sont ceux du domaine workflow, c'est-à-dire essentiellement en provenance des préceptes de management. Nous évoquions le fait qu'il existe un

grand nombre de formalismes de modélisations des workflow. La modélisation orientée processus semble être de plus en plus adoptée, en témoigne le choix du WfMC. C'est aussi la raison du choix des concepteurs de COW. Par rapport à DARE, COW, comme tous WfMS, est d'avantage orienté sur la coordination (transition – condition) et les échanges entre processus (données entrantes et sortantes). La présence de mécanismes réflexives dans COW a les mêmes origines que les autres WfMS flexibles : ils sont dus à des besoins apparus au fil du temps (cf. chapitre 1). Des politiques de management comme le BPR (Business Process Re-engineering) [SWE 95] ou le CPI (Continuous Process Improvement) [DEM 86] qui visent à effectuer respectivement des changements importants périodiquement ou des changements minimes continuellement, sont aussi à l'origine de la motivation à proposer des WfMS flexibles. Si la réflexivité est conceptuellement identifiée comme propre au support de l'activité dans DARE (*top-down*), elle est issue, pour les WfMS (et donc pour COW) de fondements pratiques (i.e. de l'expérience).

4.2.2 Le méta-modèle de COW

Le méta-modèle de COW est défini comme suit (figure 14 et figure 15).

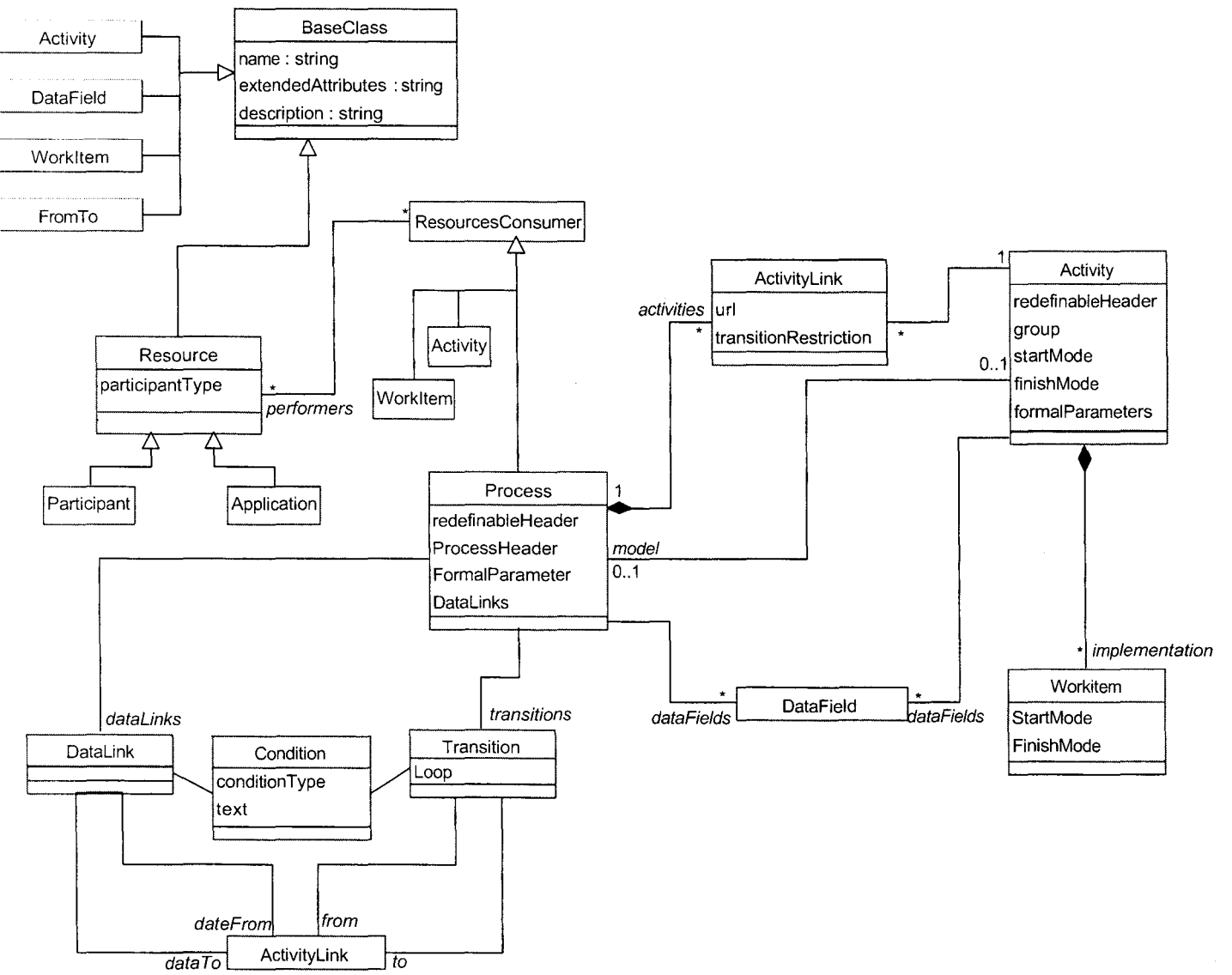


figure 14. Méta-modèle de COW (concepts de type uniquement)

Concepts de type. (cf. figure 14) Le concept *Process* correspond à notre précédent *ProcessType*. Il en va de même pour *Activity*, *Resource*, *Participant*, *Application*... Un type de processus (*Process*) est composé de type d'activité (*Activity*). Un ensemble de transitions (*Transition*) (contenant des conditions - *Condition*) régit l'ordonnancement des activités. Une activité peut être définie comme individuelle ou collective (attribut *group*). Si le type d'activité est composite, c'est qu'il s'agit d'un sous-processus du processus englobant (et non pas une activité). Dans ce cas, le type d'activité renvoie sur un type de processus (référence *model*). La réalisation d'une activité passe par celle de *workitems* (des tâches élémentaires). Ces derniers peuvent être effectués dans n'importe quel ordre. L'important est qu'ils soient tous terminés pour que l'activité puisse elle-même se terminer.

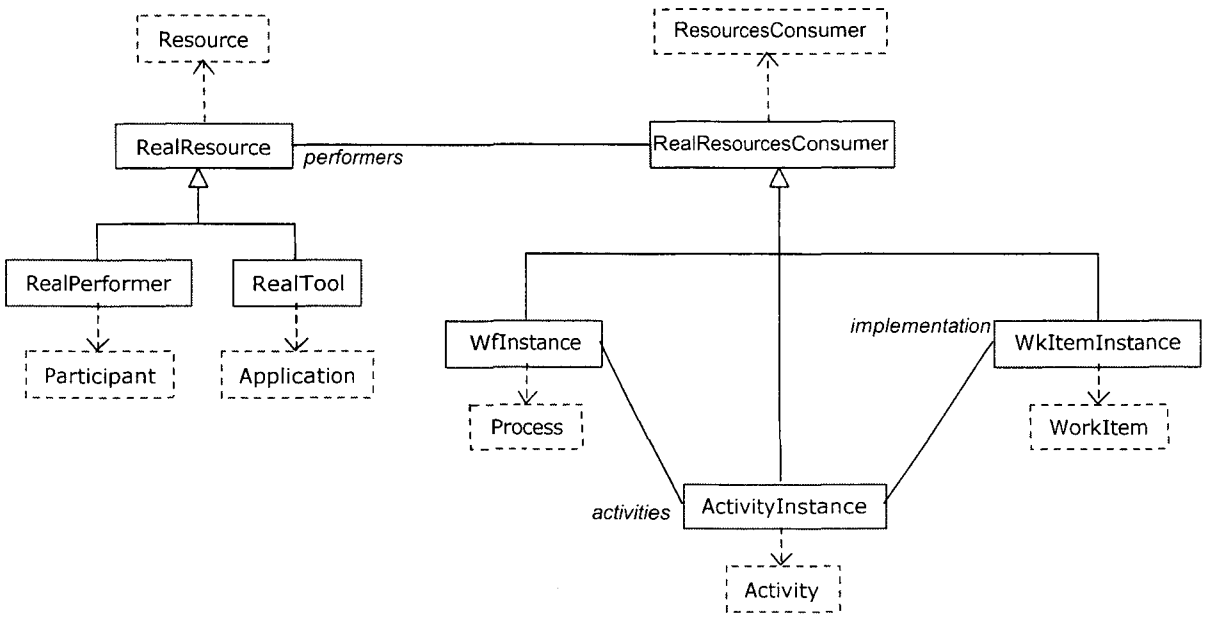


figure 15. Concepts d'instance du méta-modèle de COW

La différence des orientations des systèmes DARE (la collaboration) et COW (la coordination) n'est pas au centre de nos préoccupations et donc de notre scénario. Le scénario a juste pour objectif de mettre en scène deux méta-groupware utilisant des méta-modèles différents. Ce scénario fournit un support pour confirmer l'absence de solution d'interopérabilité entre de tels systèmes et un support pour illustrer le fonctionnement de notre solution.

4.3 Coopération de deux sociétés d'enseignement à distance (EAD)

Le contexte est composé de deux sociétés qui souhaitent coopérer. L'usage des systèmes DARE et COW est présenté dans la deuxième section. La coopération souhaitée est détaillée dans la partie 5. Le scénario qui suit est, rappelons-le, fictif.

4.3.1 Le contexte

La société HLC (Home Learning Computer) dispense des cours d'informatique. L'Espagne devenant un marché très demandeur d'informaticiens, elle décide d'inclure un cours d'espagnol. Néanmoins, elle ne dispose ni des supports de cours ni des compétences (humaines) nécessaires pour dispenser ce type de cours. HLC ne souhaite pas investir dans la mise en place de ce cours et n'en a d'ailleurs pas le temps. Pourtant elle souhaite proposer une formation qui réponde à la demande des hypothétiques étudiants : des compétences informatiques et des notions d'espagnol de manière à pouvoir travailler en Espagne. Elle s'adresse alors à la société LECS (Langues Étrangères Chez Soi) qui dispense (à distance) des cours de langues étrangères dont l'espagnol. Le cours d'espagnol fait

partie de plusieurs formations. HLC et LECS établissent une relation de partenariat : LECS va sous-traiter pour HLC en intégrant son cours d'espagnol dans une formation d'informatique.

Au premier abord, nous pouvons penser que l'intégration du module d'espagnol de LECS au sein de la formation d'HLC pourrait se résoudre à un lien au sein de la page web 'FormationInformatique' destiné à l'élève. Ce lien renverrait au site de LECS et précisément sur le module d'espagnol. Toutefois il ne s'agit alors que d'une interopérabilité de surface. Il n'y a pas de réelle intégration, de collaboration : comment imposer des contraintes temporelles (ex : le module d'espagnol doit se faire après un autre module) ? Comment vérifier que le module d'espagnol a bien été suivi ? Comment intégrer la note d'espagnol au sein de la moyenne ? Techniquement il y a deux étudiants virtuels pour le même étudiant (ce qui est déjà une erreur), sont-ils au moins liés ? ...[35] Dans notre scénario, certes hypothètique, nous visons une réelle collaboration, c'est-à-dire une interopérabilité sémantique : le module d'espagnol est intégré au sein d'HLC, et est considéré comme tous les autres modules d'informatiques (i.e. il est vu comme un module). Il est donc possible de l'associer à des règles de coordination, d'utiliser les données qu'il contient (ex : la moyenne), de vérifier pour un étudiant si le module est terminé... HLC gère ainsi l'activité globale de l'apprenant.

4.3.2 Les éléments informatiques présents

HLC intègre ce cours d'espagnol dans une formation de base. HLC utilise DARE comme système informatique. La modélisation de la formation de base est illustrée sur la figure 16. La formation de base est une tâche qui contient la sous-tâche *POO_Mod* (Module de Programmation Orientée Objet), la sous-tâche *BD_Mod* (Module de Base de Données), etc... Le module de POO contient une leçon A dont l'unique acteur sera un étudiant et l'outil sera un la page Web *CoursA*. Suit ensuite un *TP A* (une réelle tâche coopérative) dont les acteurs seront un enseignant et des étudiants (ils peuvent venir et repartir). Les outils sont l'énoncé des exercices à effectuer (*exo A*), un forum pour que l'étudiant puisse poser des questions à l'enseignant et un formulaire d'envoi pour que les étudiants puissent envoyer leurs résultats.

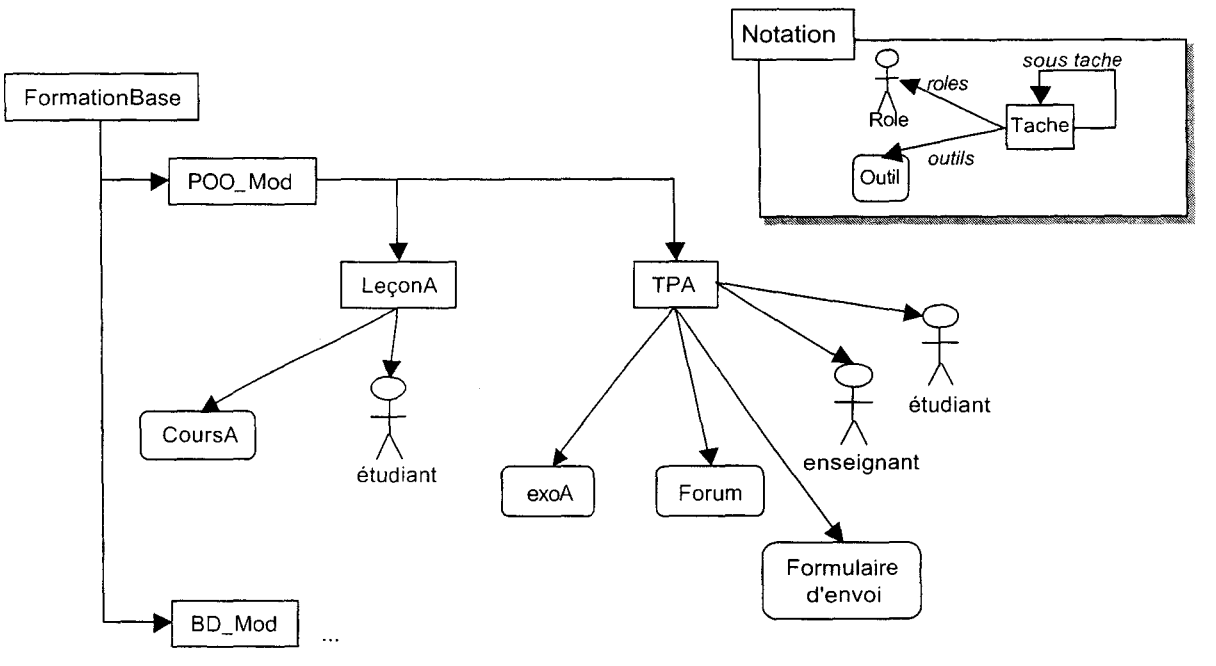


figure 16. La formation informatique de base sous DARE (HLC)

[35] Ce sont les problèmes auxquels fait face actuellement le DESS MICE EAD car chaque module est indépendant.

Le cours d'espagnol est géré par COW et se veut purement individuel. La figure 17 illustre le modèle de ce cours (il apparaît ici intégré dans une formation de base). Ce modèle a été créé par le personnel de LECS. Il s'agit d'une structure similaire aux modules précités. Le processus *Formation* est composite, il contient un cours d'espagnol et d'autres cours (non représentés). Le cours d'espagnol est lui aussi composite, il contient une leçon n°1, un TP n°1, etc. La leçon n°1 est effectuée par un étudiant à l'aide d'un outil *GramRules1*, qui est un lecteur d'un fichier Flash présentant les règles de grammaire de base.

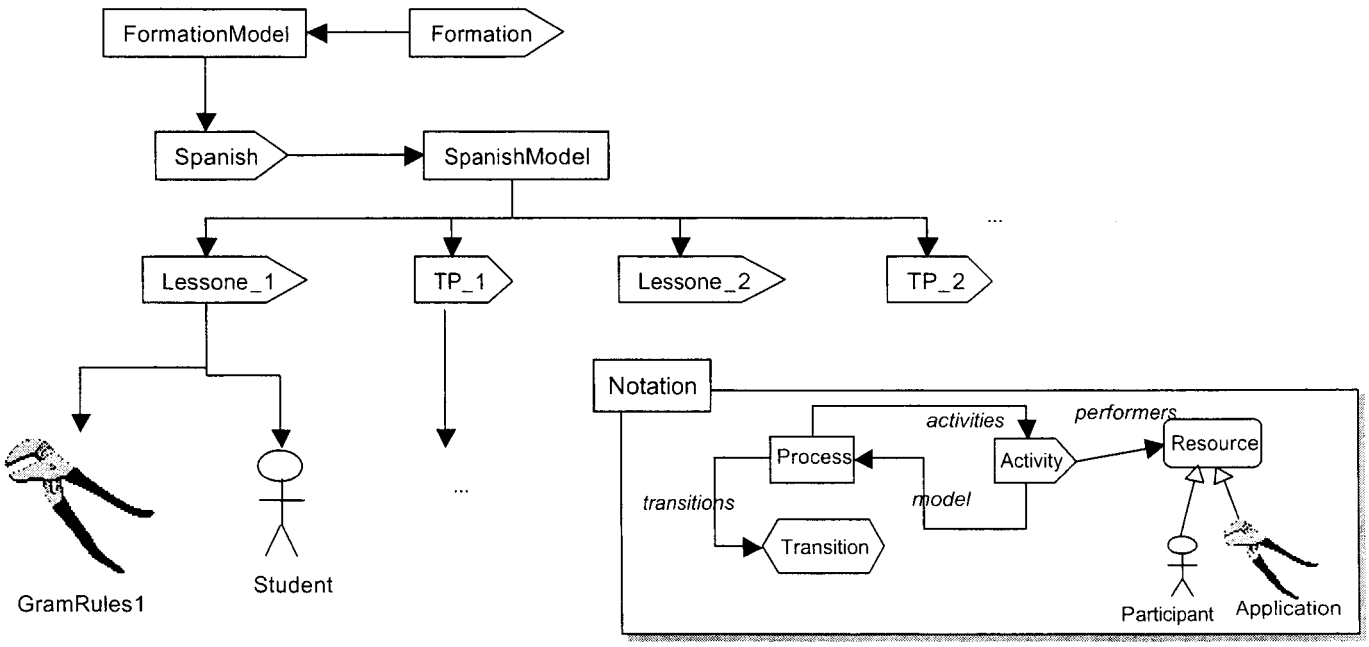


figure 17. Le cours d'espagnol dans COW (LECS)

La figure 18 montre les éléments informatiques présents. HLC utilise DARE le système dont le référentiel contient les modèles d'activités de HLC. L'exécution de DARE dans la société HLC constitue un support aux activités de cette dernière. Il en est de même pour COW qui contient les modèles de processus de LECS.

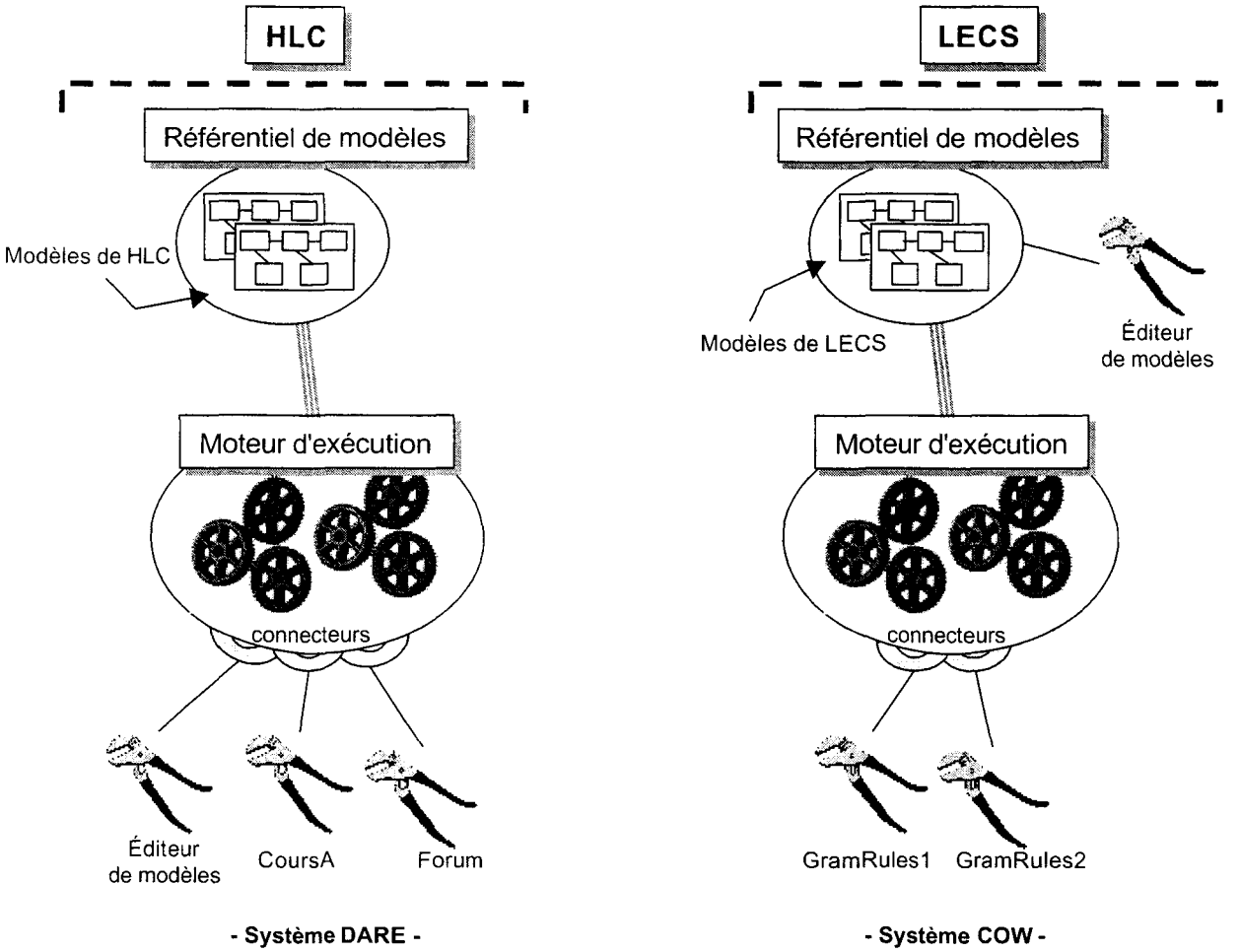


figure 18. Les systèmes DARE & COW et leur utilisation dans HLC et LECS

Notre scénario met en scène deux méta-groupware, DARE et COW, dont les bases théoriques sont différentes. Ceci a pour répercussion deux méta-modèles différents (ce qui nous intéresse). Nous avons pris deux sociétés fictives dont l'activité est l'enseignement à distance, ce qui s'inscrit dans nos préoccupations. Même si l'utilisation de DARE qui en est faite n'est pas "habituelle", elle reste cohérente et l'ensemble du scénario fournit un bon support d'étude pour vérifier l'absence de solutions d'interopérabilité adaptées à ce type de système et pour illustrer le fonctionnement de notre solution.

5 Problématique : la coopération

Cette partie est une transition entre la partie bibliographique – évolution des systèmes, importance des groupware, coopération inter-organisationnelle, les solutions d'interopérabilité et les mécanismes de flexibilité – et la partie présentant notre proposition. Sur l'exemple "concret" du précédent scénario, nous vérifions définitivement que les approches élémentaires d'interopérabilité, présentées dans le chapitre 2 sont inadaptées à la coopération de méta-groupware. La transformation de modèles étant souvent invoquée dans le contexte d'interopérabilité, nous ferons le point sur cette technique.

5.1 Les approches élémentaires

Standardisation/normalisation. Les difficultés par rapport à la normalisation d'un méta-modèle de coordination (cf. chapitre 2) restent une vérité. La spécialisation de plus en plus fréquente des WfMS, vue au début de ce chapitre, est un vecteur à la présence de plus en plus marqué des différences entre méta-modèles de systèmes workflow. De plus, notre objectif n'est pas les WfMS en

tant que tel mais les groupware en général. Il y a différentes approches pour gérer le travail collaboratif et donc implicitement différents méta-modèles. DARE en est un exemple. Le système Edubox [EDU 02] constitue un support aux activités pédagogiques. Il offre aux utilisateurs la possibilité de définir eux-mêmes leurs propres activités. Le méta-modèle intégré est dédié à la modélisation de scénarios pédagogiques. Edubox peut, par exemple, être utilisé pour supporter le travail de groupe dans un campus virtuel. Dans un tel cas, il peut être amené à coopérer avec d'autres campus, lesquels utilisant peut-être des WfMS. Les différences entre le méta-modèle d'Edubox et celui, par exemple, du WfMC sont encore plus grandes que celles présentes dans notre scénario.

Il n'est pas raisonnable de penser que la 'standardisation/normalisation' (que ce soit dans sa version 'standard fort' ou 'famille de standards') peut être une solution viable pour les problèmes d'hétérogénéité de méta-modèles. S'il n'est possible de se reposer sur un méta-modèle standard pour la coopération, il faut toutefois utiliser les standards de plus bas niveau, c'est-à-dire les supports d'implémentation des méta-modèles. Proposer une solution qui ne repose pas sur Corba, les EJB, XML, JMS ou d'autres intergiciels répandues (ou même EDI) est voué à l'échec. Si la standardisation n'est pas une réponse à notre contexte, elle en fournit néanmoins une base solide.

Les médiateurs. Une fois encore, il y a le problème de visibilité : on suppose que les sociétés LECS et HLC disposent d'une passerelle (de type médiateur) entre leur deux systèmes informatiques. Si un spécialiste de LECS modifie un des types de participants dans le module d'espagnol, il ne voit pas que celui-ci est lié avec le rôle correspondant dans le système de HLC. Pour cela, il devrait utiliser l'éditeur du médiateur SI celui-ci existe. Et comme chaque élément d'un modèle est sujet à des liaisons "externes", l'éditeur de COW devrait être abandonné au profit de celui du médiateur. Une fois encore, le degré d'autonomie de cette solution est faible. La situation se complique avec plusieurs coopérations, concernant à chaque fois un système différent. Le médiateur se doit de gérer toutes les coopérations, sinon il peut y avoir des incohérences. Cette universalité implique que tous les systèmes utilisent le même type de médiateurs. C'est encore une exigence forte.

Pour éviter le problème de la visibilité, le report de toute modification de manière automatique dans le référentiel du médiateur suivi de la répercussion par celui-ci dans les autres systèmes est une possibilité. Cela oblige néanmoins le système à implémenter et à intégrer un lien fort avec le médiateur (comme une couche d'événement). Ceci diminue aussi le degré d'autonomie. Nous verrons plus en détail cette solution dans la section suivante.

L'approche par les médiateurs reste toutefois séduisante. Le fait d'externaliser l'implémentation de la coopération dans un composant tiers reste la méthode la plus efficace vis-à-vis du degré d'autonomie.

Les interactions par spécifications. Cette solution n'est pas non plus adaptée. Une coopération automatique (objectif de cette solution) nécessite la description d'informations de haut niveau telles que les nécessités, les objectifs des processus (d'un point de vue commercial). Le développement d'un méta-modèle avec des concepts très abstraits pour écrire ce genre de descriptions est une possibilité pour la coopération. Néanmoins, nous y voyons deux inconvénients :

- Ces descriptions sont complexes. Étant à fournir par les utilisateurs, cela est difficilement exigible. Même si elles sont réalisables, elles demandent dans ce cas beaucoup d'efforts. Chaque modification sera coûteuse.
- Chaque système doit utiliser le (même) méta-modèle[36]. Une telle obligation revient à une standardisation (sémantique).

5.2 Le point sur la transformation de modèles

Actuellement, la transformation de modèles est souvent évoquée pour les questions en rapport avec l'interopérabilité. Qu'en est-il dans notre contexte ?

[36] Ou les concepts additionnels "abstraits" qui s'ajouteraient à chaque méta-modèle.

La transformation de modèles est une opération qui consiste à traduire un modèle issu d'un méta-modèle A en un modèle issu d'un méta-modèle B. La transformation intervient dans l'approche par médiateurs : les solutions d'EAI, le data warehouse avec la transition vers le langage pivot. Elle intervient aussi dans la conception d'applications : un modèle d'analyse très élaboré est traduit dans un modèle orienté conception/implémentation [BOU 00b], un modèle indépendant de la plate-forme est traduit en un modèle dépendant d'une plate-forme [RAY 01]. La transformation est généralement utilisée en mode statique : des échanges de spécifications entre ateliers de conception utilisant des méta-modèles (de conception) différents.

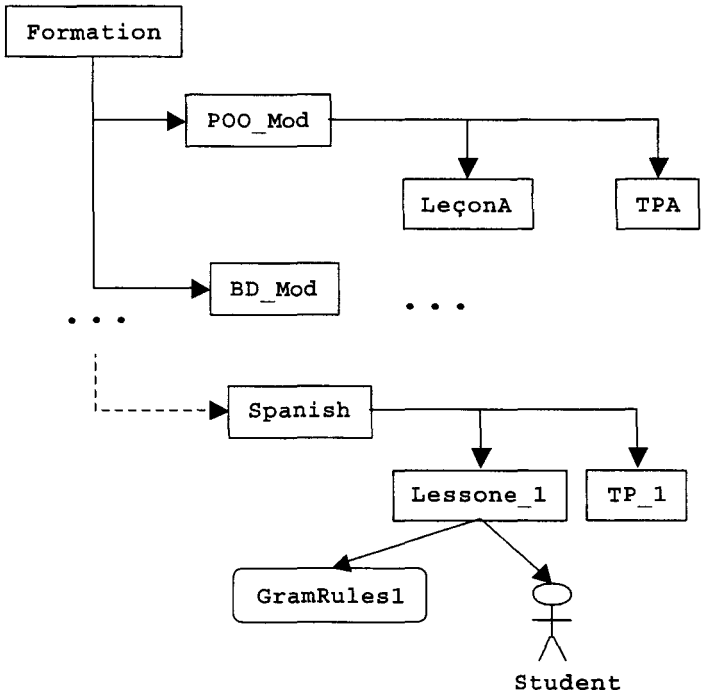


figure 19. Le module d'espagnol exprimé dans DARE

Sur notre exemple (cf. figure 19), l'utilisation la transformation de modèles pour supprimer la différence de méta-modèle pourrait sembler être une solution. Si le modèle de LECS est transformé comme modèle "DARE" et intégré dans le système d'HLC, les utilisateurs de celui-ci peuvent lier *Spanish* à *Formation*. Néanmoins, à l'exécution apparaissent plusieurs problèmes.

- Lorsque le type du processus suivant est *Spanish* (issu d'une transformation), le système de HLC doit créer le processus dans le système de LECS. La **situation géographique** est la première difficulté. L'instanciation du type doit se faire dans l'autre système. La transformation doit contenir des informations indiquant où se trouve le modèle d'origine. Le système doit savoir utiliser ces informations : c'est-à-dire se connecter à ce modèle et invoquer l'opération correspondant à l'instanciation (problème de l'accès informatique).
- Des différences de **logique de contrôle** sont ensuite à gérer. Le système de LECS exécute un processus *Spanish* créé par HLC. Lorsque celui-ci est terminé, comment le moteur d'exécution de HLC récupère la main ? Doit-il placer un adaptateur Activity→WfInstance qui recevra les informations sortantes de précédent processus ? Doit-il s'inscrire en tant qu'auditeur à un canal d'événements ? ...
- Les **différences de protocoles d'échanges** de données entre processus (envoie et réception) doivent être supprimés. Les processus de COW n'échangent sûrement pas de la même manière les données entre eux telles que le font les activités de DARE.

Dans notre contexte, la transformation de modèles n'est pas une réponse à la coopération. D'autres mécanismes doivent lui être associés pour fournir une solution. Toutefois deux aspects de la transformation sont problématiques dans un contexte dynamique : les changements et les valeurs dérivées.

Changements. Quand un utilisateur de LECS effectue un changement dans le modèle, il faut re-générer la transformation pour le système de HLC, et donc remplacer l'ancien modèle par le nouveau. Pour les éléments inchangés, il faut toutefois transposer les liens sur le nouveau modèle. La régénération et la transposition sont les deux problèmes impliqués par les changements dynamiques.

Trois mécanismes additionnels peuvent permettre de supporter les changements dans les transformations : 1) une couche d'événement pour automatiser les régénérations 2) un mécanisme d'indication des types d'éléments et des caractéristiques modifiés 3) un modèle de transformations (accessible pendant l'exécution) qui indique les relations de transformations entre types et caractéristiques. Ainsi quand une modification intervient, la répercussion est automatiquement faite sur le modèle du coopérant de manière localisée. Ces mécanismes appartiennent à la structure de transformation des spécifications CWM (Common Warehouse Managment) de l'OMG, élément central de sa nouvelle *architecture conduit par les modèles* (MDA).

Valeurs dérivées. Certaines caractéristiques ont des valeurs qui sont dépendantes du temps, c'est-à-dire qui sont calculées à chaque consultation : par exemple un attribut qui renvoie l'heure. La transformation représente une copie à l'instant t, et donc quelques instants après, ne représente plus la réalité. Dans notre contexte, les modèles "transformés" doivent être vivants : si les liaisons de coopération sont basées sur quelque chose de provisoire ou d'éphémère, qui n'est pas à jour, la synchronisation échoue. Des références sur des éléments qui n'existent plus peuvent générer des erreurs. De telles erreurs peuvent être en dehors de l'ensemble des exceptions gérées par le système (dont la conception ne prévoyait pas de coopération).

Si la transformation pose des problèmes ou nécessite des mécanismes additionnels, adapter un modèle à un autre système reste pourtant un élément inévitable dans une solution de coopération. Dans notre proposition nous adoptons une technique dérivée de la transformation.

6 Conclusion

Fournir un support informatique à la coopération d'entreprises revient à faire coopérer les groupware utilisés. Ces systèmes constituent le support aux organisations (coordination, communication, coopération). Une base importante de ces systèmes est issue des WfMS. Nos préoccupations concernées dans le chapitre les systèmes dit "flexibles". Ils semblent que la structure la plus puissante et la plus prometteuse pour la flexibilité des groupeware soit celle utilisée dans l'approche par méta-modèles : méta-modèle, structure à trois niveaux et réflexivité. Le problème logique et principal est que les groupware n'utilisent pas (ne peuvent pas utiliser) tous le même méta-modèle ou le même accès informatique. Sur l'exemple de deux sociétés d'EAD nous avons montré que les approches habituelles n'apportent de support pour la coopération à ce type de systèmes.

Les méta-groupware, systèmes qui seront à notre avis de plus en plus adoptés par le monde industriel, n'ont pour l'instant aucun outil/technique/approche pour coopérer entre eux. Face à ce manque, nous proposons une solution.

Chapitre 4

CAST

Un outil pour la création de services d'adaptation

Les solutions d'interopérabilité actuelles ne sont adaptées aux contraintes impliquées par la flexibilité. La normalisation (ici la normalisation du méta-modèle et de son support informatique) est un processus rarement couronné d'un succès total. Les médiateurs (data warehouse, fédération de BD, solutions d'EAI...) offrent une coopération, gérée par un tiers, qui n'apparaît pas dans les systèmes. Et enfin les ontologies sont trop complexes pour les utilisateurs. Le principal inconvénient de ces solutions est surtout de ne pas prendre en compte la structuration en trois niveaux. Aucune ne profite des liens forts entre chaque niveau : elles ne sont donc pas adaptées à l'approche par méta-modèles.

Les travaux autour de la méta-modélisation sont limités à la transformation de modèles. Cette opération, qui peut intervenir dans des solutions d'interopérabilité (data warehouse), n'est malheureusement pas une technique adaptée à un contexte dynamique : si des liens sont faits avec un modèle issu d'une transformation, et que le modèle source a changé, les liens ne sont plus bons.

Devant l'absence de réponse à ce problème de coopération, il nous est apparu important d'apporter une solution. Nous proposons une solution (CAST - *Creation of Adaptation Service Tool*) basée sur la notion d'adaptateur : les modèles contenus dans un référentiel vont pouvoir être vus et référencés par d'autres (référentiels de) systèmes après avoir été adaptés. L'adaptation consiste en une sur-couche qui traduit chaque élément en son équivalent. Pour que deux modèles hétérogènes puissent être "liés", chaque modèle est adapté à l'autre. Encore une fois, nous insistons sur le fait que notre étude porte sur le dénominateur commun aux méta-groupware : l'approche par méta-modèles. CAST n'intègre pas de concepts spécifiques aux systèmes de workflow ou à l'approche de DARE (coordination légère). CAST doit être suffisament générique pour faire coopérer tout type de méta-groupware, ce qui est bien l'objectif initial de nos travaux.

Dans un premier temps, nous exposons le fonctionnement général de CAST après avoir expliqué au préalable notre démarche. L'utilisation de CAST se fait en deux phases : 1) établissement des liens d'adaptation entre les méta-modèles par les concepteurs et 2) liaison des modèles par les utilisateurs grâce au service d'adaptation généré grâce aux premiers liens. La partie 2 décrit le formalisme de liaison de CAST pour les liens entre méta-modèles (phase 1). L'objet de la partie 3 est le fonctionnement des services d'adaptation (phase 2) et plus précisément le principe et la structure des adaptateurs mis en œuvre. Dans ce chapitre, nous ne nous soucions pas du support informatique de CAST (i.e. de sa réalisation pratique). La partie 4 décrit le modèle conceptuel d'un service d'adaptation, c'est-à-dire indépendamment d'une plateforme particulière.

1 Fonctionnement général

CAST (*Creation of Adaptation Service Tool*) est la solution que nous avons conçue pour la coopération inter-organisationnelle. Il s'agit d'un environnement qui permet de générer des services d'adaptation entre systèmes. Cette partie débute par la description de notre approche (1.1) : les caractéristiques que nous reprenons des approches élémentaires d'interopérabilité avec les modifications et les ajouts que nous y apportons par rapport à nos préoccupations, le principe général de fonctionnement (et sa réponse aux différentes contraintes énoncées jusqu'ici) et l'illustration de ce dernier sur l'exemple du scénario "HLC-LECS". Nous détaillons ensuite la manière avec laquelle CAST est utilisé (1.2) et les éléments employés ou créés pendant cette utilisation (1.3).

1.1 Notre approche

1.1.1 Éléments utiles des solutions d'interopérabilité existantes

La normalisation, l'approche par médiateurs et la transformation de modèles n'apportent pas de réponse à notre contexte. Néanmoins elles fournissent une base à celui-ci. Il n'est pas possible de se reposer sur un méta-modèle standard (avec un accès informatique standard) pour notre problématique. Pourtant il est obligatoire de se reposer sur un ou plusieurs standards de plus bas niveau, c'est-à-dire un support d'échanges de données ou d'invocation de services. Les normes comme XML, Corba, WSDL/SOAP, le bus de messages de MQSeries sont des exemples de ces standards potentiels. Utiliser des médiateurs de la manière habituelle pose des problèmes de visibilité (implication des utilisateurs) et de report de modification (contexte dynamique). La transformation, qui est un des éléments clés d'un grand nombre de solutions orientées médiateur, est peut-être l'élément inadapté au contexte dynamique, que ce soit pour les valeurs dérivées ou les régénérations. Malgré cela, externaliser le composant de coopération et adapter les modèles aux systèmes coopérants semblent inévitables.

Notre proposition reprend ces trois éléments (la normalisation, l'approche par médiateurs et la transformation de modèles) avec un degré d'utilisation particulier ou en y apportant nos propres variations pour les adapter à la coopération. Quatre aspects constituent notre proposition :

- Utilisation d'un standard pour l'accès aux données et aux fonctionnalités.
- Utilisation d'un standard pour l'accès informatique des méta-modèles.
- Placer un médiateur entre les systèmes, spécifique à ces derniers.
- Remplacer les règles de transformation par des règles d'adaptation de manière à créer des adaptateurs plus appropriés dans un contexte dynamique. Le médiateur a pour but de créer et gérer ces adaptateurs.

Ce chapitre a pour but de décrire le médiateur, que nous appellerons *service d'adaptation* (AS), d'un point de vue conceptuel. Son application à des standards de "données" et un accès informatique particulier (des méta-modèles) sera, par contre, abordée dans les deux chapitres suivants.

1.1.2 L'idée principale

La coopération entre systèmes commence par la liaison des modèles de fonctionnement. Par rapport aux contraintes de visibilité et d'accessibilité, la solution la plus efficace est la définition des liens entre modèles par les utilisateurs-spécialistes directement à partir de l'éditeur de leur système. Notre proposition consiste d'abord à placer des adaptateurs sur un référentiel de modèle de manière à adapter les modèles qu'il contient à d'autres référentiels de nature différente. Des utilisateurs d'un autre système peuvent donc lire ces modèles 'externes' et lier des éléments de ceux-ci à des éléments de leurs modèles. Ces liaisons sont des affectations de valeurs aux propriétés des précédents éléments. Les modèles externes ont été modifiés. Ces modifications doivent apparaître dans leur système d'origine. Pour cela, les adaptateurs permettent d'affecter des valeurs : ils traduisent les requêtes d'affectation. Dans le système d'origine, le modèle est modifié et se voit lié avec des éléments

externes. C'est au tour de ces éléments d'être adaptés au système d'origine. Il y a donc une double adaptation.

Par rapport à la transformation, l'adaptation ne rencontre pas le problème de *situation géographique* : la requête d'instanciation est traduite et envoyée au type réel. Le problème des *changements* est lui aussi inexistant : chaque consultation d'une propriété consiste à chercher l'information à la source et à la traduire. Un élément adapté est toujours à jour. Cette caractéristique supprime également le problème des *valeurs dérivées.*

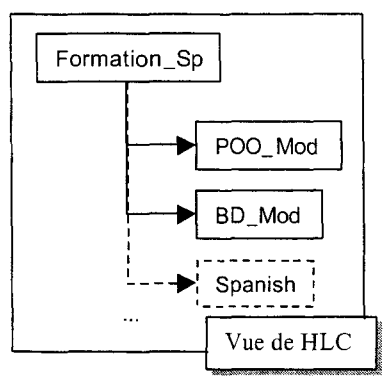
Les différences de *logique de contrôle* et de *protocole d'échanges* interviennent au niveau M-zéro, c'est-à-dire, concernent les instances de processus ou similaires. Pour supprimer ces différences, nous utilisons encore une fois des adaptateurs. Toutefois en utilisant au maximum les liens d'instanciation présents dans la structure à trois niveaux, l'architecture de ces adaptateurs "M-zéro" se déduit en partie des concepts de type. **C'est un apport important de notre proposition : utiliser la particularité de la structure à trois niveaux.**

Quelle est la place de CAST par rapport à l'utilisation d'adaptateurs ? La notion d'adaptateur est très présente dans le domaine informatique. Pour la programmation [GoF 95][KRU 97][LAV 96], un adaptateur permet de ré-utiliser des fonctionnalités en ne modifiant que les points d'accès de celles-ci pour les adapter à un autre programme. Dans les couches réseaux bas niveaux, les adaptateurs permettent de supprimer certaines différences de protocoles [KAN 88][WIL 86]. Des intergiciels comme Corba utilisent des adaptateurs pour masquer les différences d'implémentation des objets présents dans le même environnement [VIN 98][DOG 98]. Enfin le partage de ressources par plusieurs systèmes est parfois géré par des adaptateurs qui adaptent l'accès à la ressource pour le système et peuvent aussi appliquer une politique concernant l'accès (concurrence) [MAN 99][TRE 94]. Dans ces exemples, les adaptateurs sont implémentés de manière ad-hoc : les développeurs les conçoivent pour un contexte donné. Notre idée est de factoriser au maximum la structure et le comportement des adaptateurs pour le contexte de coopération de méta-groupware. Le principe de CAST est d'être un outil qui permet de développer, pour deux systèmes différents, un service d'adaptation le plus rapidement possible. Le service adapte les modèles et leurs instances pour qu'ils puissent être utilisés par le "coopérant". Le gain de temps est réalisé grâce à un formalisme de définition de règles d'adaptation, à la déduction de caractéristiques des éléments M-zéro, à la présence de fonctions d'adaptation déjà implémentées et surtout à un moteur évolué de création et de gestion d'adaptateurs qui évite les problèmes de typage, de nommage et de duplications inutiles (problèmes étudiés dans la section 3.2).

1.1.3 La coopération souhaitée sur l'exemple du scénario HLC - LECS

Nous donnons ici un aperçu du fonctionnement du service d'adaptation "généré" pour l'exemple du scénario HLC - LECS. Même s'il y a des différences, entre DARE et COW, de fonctionnement (sur lesquelles nous ré-insistons), l'aperçu montre que le service d'adaptation (spécifique à DARE et COW) permet une réelle coopération des systèmes, coopération définie de surcroît par les spécialistes.

HLC souhaite intégrer le cours d'espagnol de LECS dans une de ses formations. Pour cela, un service d'adaptation entre les deux va être mis en place. HLC indique que la nouvelle tâche *Formation_Sp* (Sp pour spanish) est visible pour LECS et celui-ci indique que *Spanish* est visible pour HLC. Le spécialiste de HLC va voir *Spanish* dans son modeleur (en important explicitement le contenu de LECS) sous la forme d'une tâche et va venir l'insérer dans *Formation_Sp.*



Lorsqu'un étudiant suit la *Formation_Sp* il suit chaque cours sur son navigateur Web. Il verra l'interface habituelle pour les cours informatiques mais en verra une autre pour le cours d'espagnol. Il sera automatiquement redirigé vers le site de LECS. Il faut rappeler que la gestion de la coordination du système de HLC (i.e. DARE) est moins rigoureuse que celle du système de LECS (i.e. COW) : dans la partie gérée par HLC, l'étudiant peut choisir à tout moment de rentrer dans un module (ex : POO) et de prendre n'importe quel cours ou TP (ex : TP_C).

L'avantage du système utilisé est son aspect coopératif dont les possibilités sont très évoluées. Il peut coopérer facilement lors d'un TP. Dans la partie gérée par LECS, l'étudiant effectue les tâches de manière individuelle (l'aspect coopératif des activités est moins puissant) par contre la coordination des cours et des TPs est fine et stricte l'étudiant peut commencer chaque module parallèlement, mais à l'intérieur de ceux-ci il doit suivre un ordre précis : Cours n°1, TP n°1, Cours n°2, TP n°2... L'étudiant qui suit une *formation_SP* voit une liste de modules possibles (DARE), s'il choisit le module *POO* il va où il veut. Par contre, s'il choisit le module *Spanish* (COW), il sera obligé de reprendre là où il avait arrêté (à la dernière utilisation du module *Spanish*) et de suivre l'ordre obligatoire. Pour la fin d'un module, c'est l'étudiant qui indique qu'il souhaite arrêter (sans indiquer s'il a terminé ou non chaque cours et TP) dans le cas de HLC. Dans le cas de LECS, l'étudiant doit remplir certaines conditions pour terminer un TP ou un cours. Lorsqu'il s'agit du dernier "processus" (cours ou TP) du module, ce dernier est alors fini. Pour terminer la *formation_SP*, l'étudiant envoie un message au tuteur qui vérifie que rien n'a été oublié (tous les modules sont-ils terminés ?).

1.2 Mode d'emploi de CAST

Le principe d'utilisation de CAST est illustré sur la figure 20. CAST propose un environnement pour permettre d'établir les correspondances entre éléments des méta-modèles. Ces liaisons se font entre concepts de types (pour les modèles) et entre concepts d'instances (pour les instances de modèles). Ces équivalences sont effectuées par un informaticien qui connaît les deux méta-modèles [étape 1][37]. Lorsque celles-ci sont définies, CAST génère un service d'adaptation qui sera connecté aux deux systèmes par les utilisateurs. Ainsi les utilisateurs de A pourront voir les modèles de B dans leur système (car traduits dans leur formalisme de modélisation). Ce service permettra donc aux utilisateurs de A ou de B de tisser des liens avec des éléments extérieurs [étape 2]. Les liaisons définies constituent le *modèle inter-organisationnel.* Ensuite les instances de modèles, processus, tâches, ou autres sont créés et exécutés sur le système adéquat. Ces instances se synchronisent grâce au modèle inter-organisationnel et peuvent s'échanger des informations grâce aux adaptateurs M-zéro qui suppriment les différences de protocole d'échange (et de logique de contrôle) [étape 3].

[37] Un de nos objectifs prochains est d'avoir un environnement coopératif et suffisamment simple pour que ce soient les utilisateurs des systèmes qui puissent effectuer les équivalences.

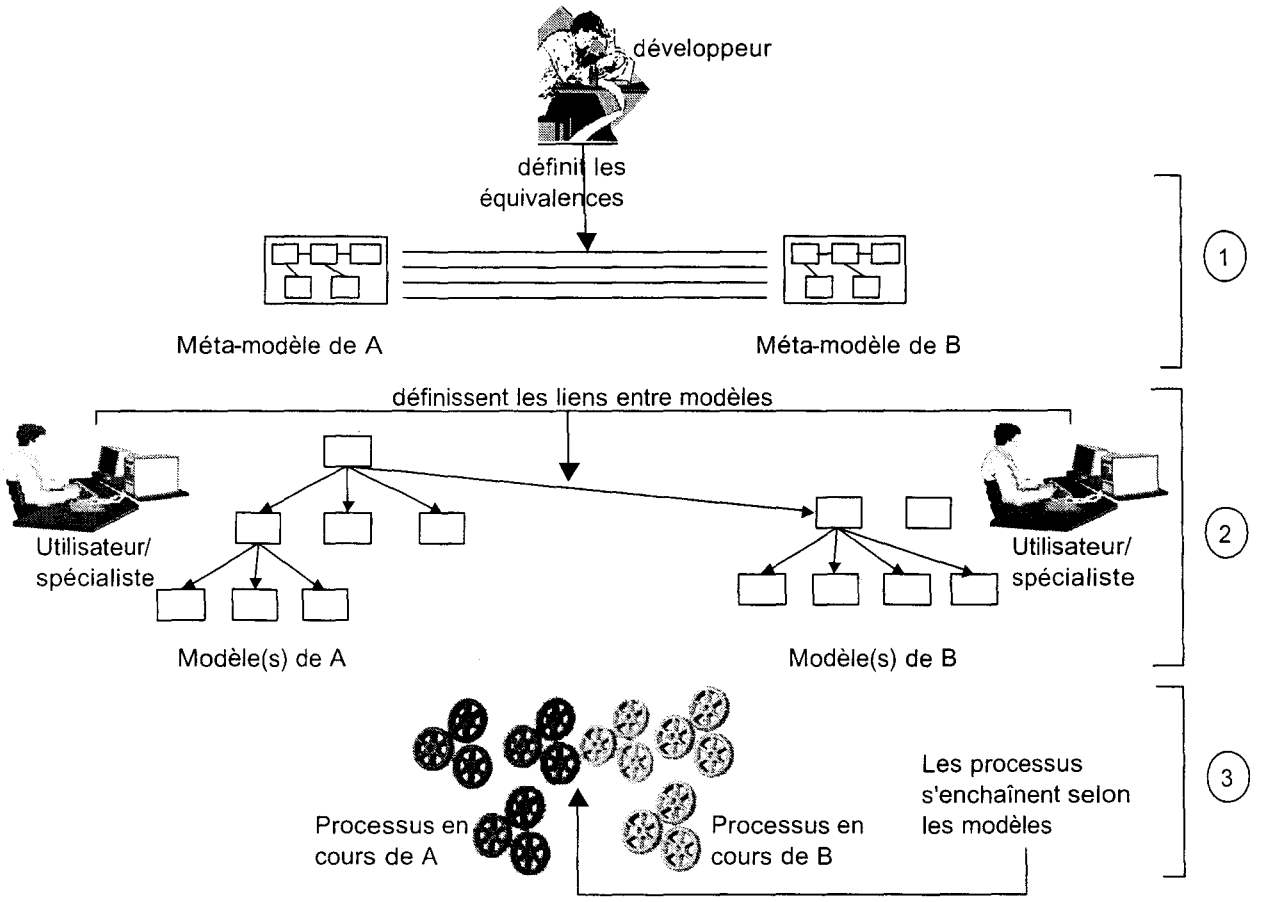


figure 20. Utilisation de CAST

1.3 Principe de fonctionnement

Les liaisons d'adaptation entre les deux méta-modèles sont envoyées au premier élément de CAST : le générateur de services (cf. figure 21). Celui-ci construit le service d'adaptation spécifique aux deux méta-modèles. Une fois connecté aux deux systèmes, le service crée un ou des adaptateurs seulement à la demande d'un des deux systèmes. Par exemple, si un employé d'HLC demande de voir toutes les tâches de LECS, le service d'adaptation crée un adaptateur pour chaque type processus du système LECS. L'employé peut aussi chercher la tâche *Spanish*, dans ce cas le service ne crée qu'un adaptateur (si la recherche est un succès). Ces recherches sont bien sûr des fonctionnalités proposées par le référentiel HLC (donc de DARE) dont la traduction vers les fonctions similaires fait partie des règles d'adaptation.

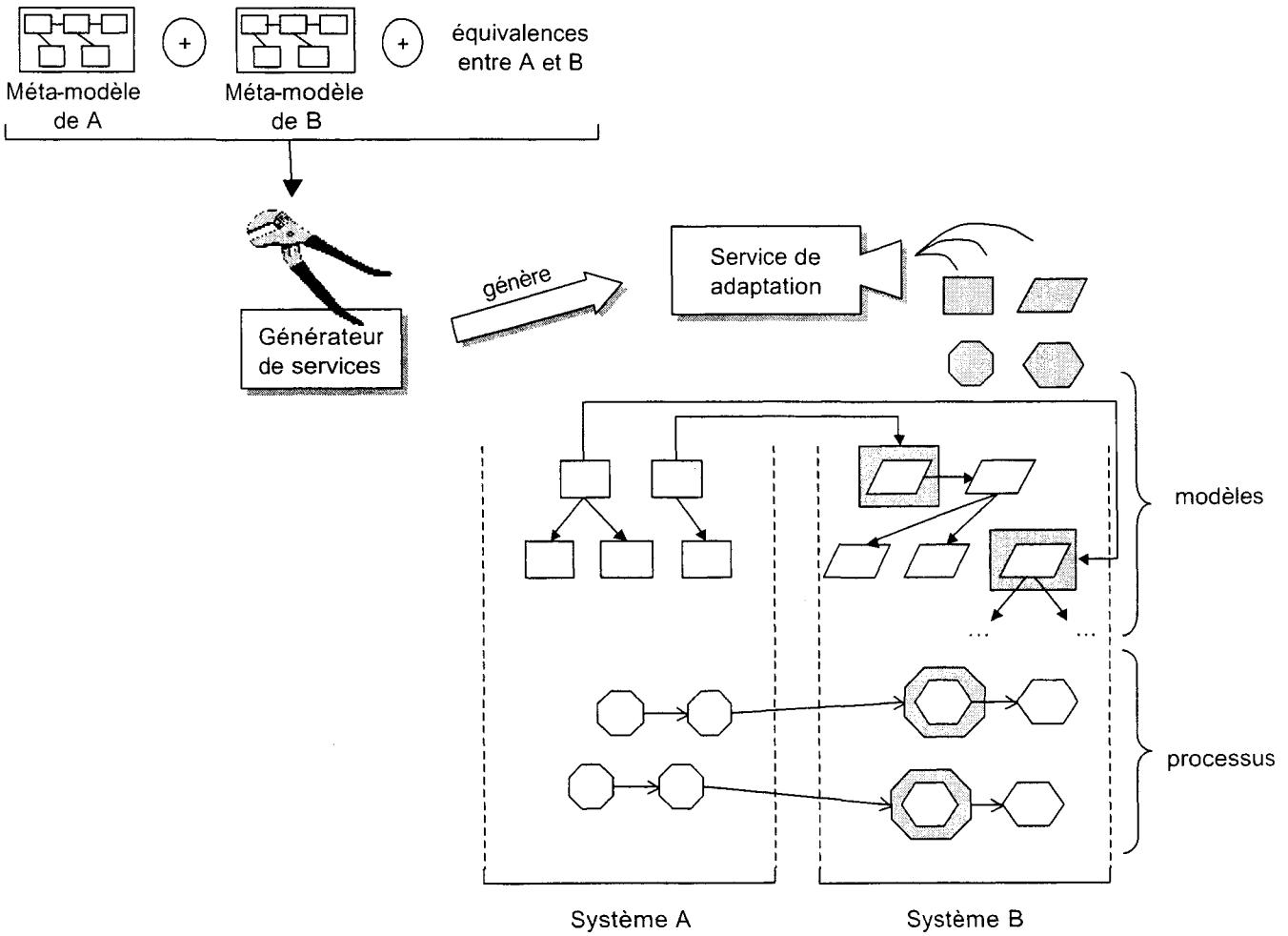


figure 21. Un service d'adaptation

Lorsqu'une instance de *Formation_SP* "s'exécute", elle est amenée à lancer un cours d'espagnol. Pour cela, elle demande au type de processus *Spanish* de créer une instance. Plus précisément, elle envoie une requête d'instanciation à *Spanish'*, l'objet adaptateur d'*Spanish*. L'adaptateur transmet (après traduction) la requête à *Spanish* qui crée et renvoie un processus *P*. *Spanish'* renvoie *P* encapsulé dans une activité *P'* de type *Spanish'* qui constitue l'adaptateur de P. L'instance de *Formation_SP* peut transmettre les informations à *P'* et démarrer celui-ci.

La structure de chaque type d'adaptateurs est créée à partir des liens d'adaptations entre méta-modèles. C'est l'objet de la partie suivante.

2 Liaisons d'équivalence

Note. La notation X(Y) indique qu'il s'agit d'un adaptateur de type X qui adapte un élément de type Y. Par exemple, Spanish' est un Task(Process) ➔ *Spanish' adapte les types de processus (COW) en tâche (DARE). La notation X[y] indique qu'il s'agit d'un élément qui adapte y (de type non précisé) en un élément de type X. Ex : Spanish'=Task[Spanish].*

La figure 20 indique que la première étape consiste à établir les liens d'équivalence entre les concepts de type et d'instance des deux méta-modèles. Dans le reste de ce chapitre, nous ne nous

attarderons pas sur le formalisme de méta-modélisation utilisé. Le formalisme de liaison d'adaptation (de CAST) présenté ici se veut indépendant de tout formalisme et de tout support d'exécution dans le but de pouvoir s'appliquer à tout type de formalisme et tout support d'exécution. Dans le chapitre 5, il sera utilisé tel quel sur le formalisme de méta-modélisation MOF. Le paradigme objet reste toutefois privilégié pour sa modularité et sa place de plus en plus forte dans la modélisation. Le formalisme UML et le langage Java sont respectivement employés pour décrire les concepts utilisés dans les exemples et exprimer des fonctions ou des contraintes. Le formalisme d'expression des règles d'adaptation de CAST peut être utilisé avec d'autres formalismes et langages de programmation.

2.1 Principe général

Le principe de fonctionnement d'un AS est de créer une surcouche, c'est-à-dire un adaptateur, pour chaque élément de manière à l'intégrer dans un autre système (plus précisément un autre référentiel) (cf figure 22). Lorsque qu'un adaptateur reçoit une requête [étape 1], il la traduit en une requête de l'élément adapté, c'est-à-dire la source [étape 2]. La réponse à cette requête est traduite par l'adaptateur en élément(s) de B. Dans la traduction il y a une partie (éventuelle) de traitement supplémentaire à effectuer [étape 3], et une partie adaptation d'une valeur d'un type en un autre type [étape 4].

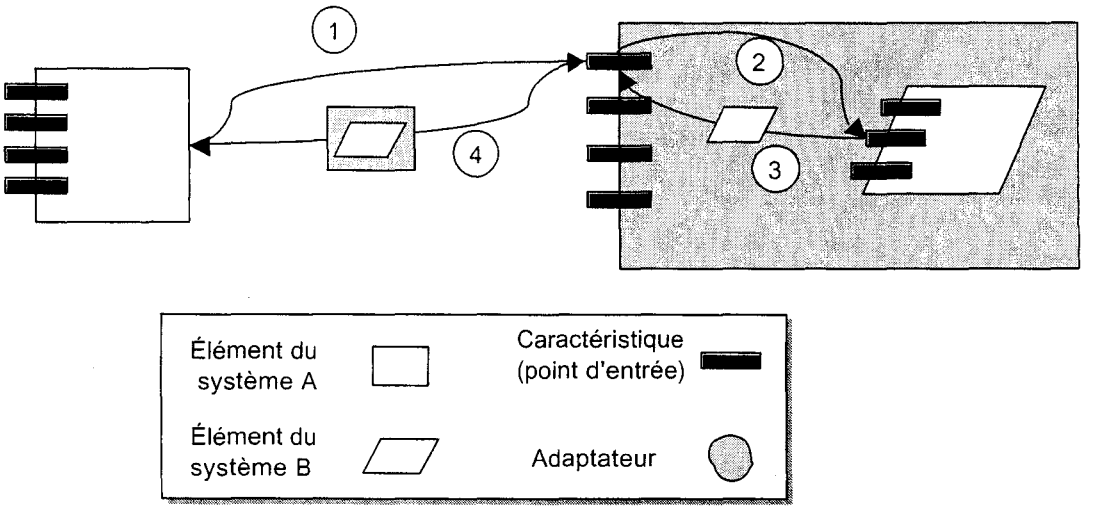


figure 22. Étapes d'adaptation

La figure 23 illustre ce fonctionnement sur l'exemple HLC-LECS. Si la tâche *Formation_SP* demande à sa sous-tâche Task[*Spanish*] quelle est le premier type d'outil utilisé [étape 1], la traduction de la requête consiste à obtenir les ressources référencées à l'aide de *performers* [étape 2] et à prendre la première application [étape 3]. *GramRules1*, cette application, est ensuite adaptée pour DARE : Tool[*GramRules1*].

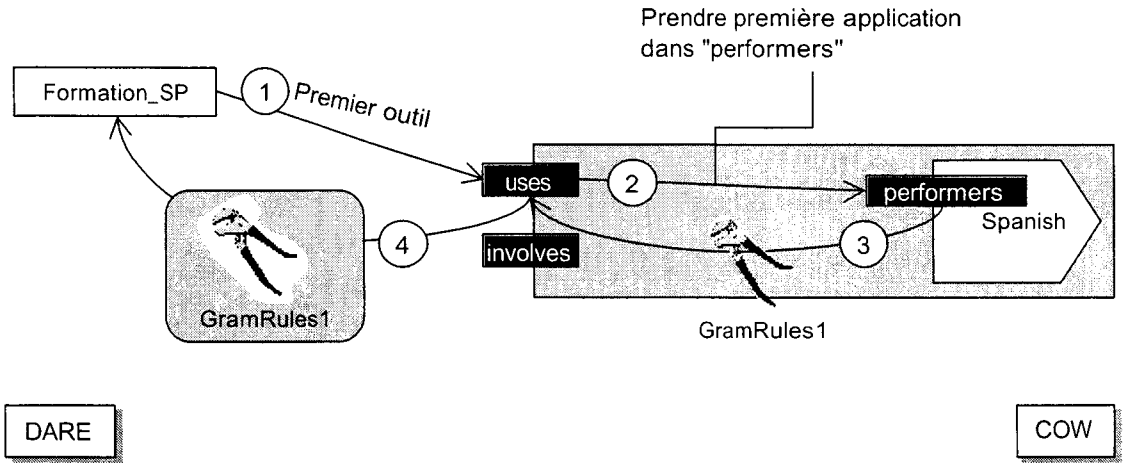


figure 23. Fonctionnement des adaptateurs sur l'exemple

Un des enjeux de CAST est de factoriser le plus de code d'implémentation possible, c'est-à-dire l'implémentation commune à tous les services d'adaptation, quelque soit les méta-modèles. Un autre objectif est de proposer des accélérateurs d'implémentation à travers des artefacts graphiques. La factorisation est étudiée dans les parties 3 et 4. Le formalisme des règles d'adaptation (associé à CAST), composé des artefacts graphiques d'accélération, est l'objet de cette partie. Les règles indiquent comment une requête est traduite en une autre requête, et en quel type d'éléments, un élément peut être adapté. Le premier point correspond aux liaisons structurelles[38] et le deuxième aux liaisons conceptuelles.

2.2 Sources d'inspiration

Les origines du formalisme décrit dans les sections suivantes sont diverses. Nous nous sommes inspirés pour les artefacts graphiques des éditeurs de transformation présents dans les solutions d'EAI, principalement BizTalk Mapper.

En plus de proposer le support de nombreux formats (et les transformations entre eux), BizTalk Server [BIZ 02] propose donc un éditeur pour créer ses propres règles de transformation : BizTalk Mapper (cf. figure 24). Il a pour but de traduire des messages XML en un format A en message de format B. Le moteur de transformation est basé sur XSLT. Le but de l'éditeur est de faciliter la création de règles XSLT. Celui-ci génère ensuite les règles XSLT correspondante. L'éditeur propose de tisser des liens entre balises XML (n → 1), d'y associer des fonctions (appelées *Functoids*).

[38] L'adjectif structurelle n'est pas ici en opposition avec comportementale. Dans son association avec liaison, il englobe les liaisons des caractéristiques structurelles (attribut, référence, ...) et comportementales (opérations, fonctions, ...) entre deux concepts.

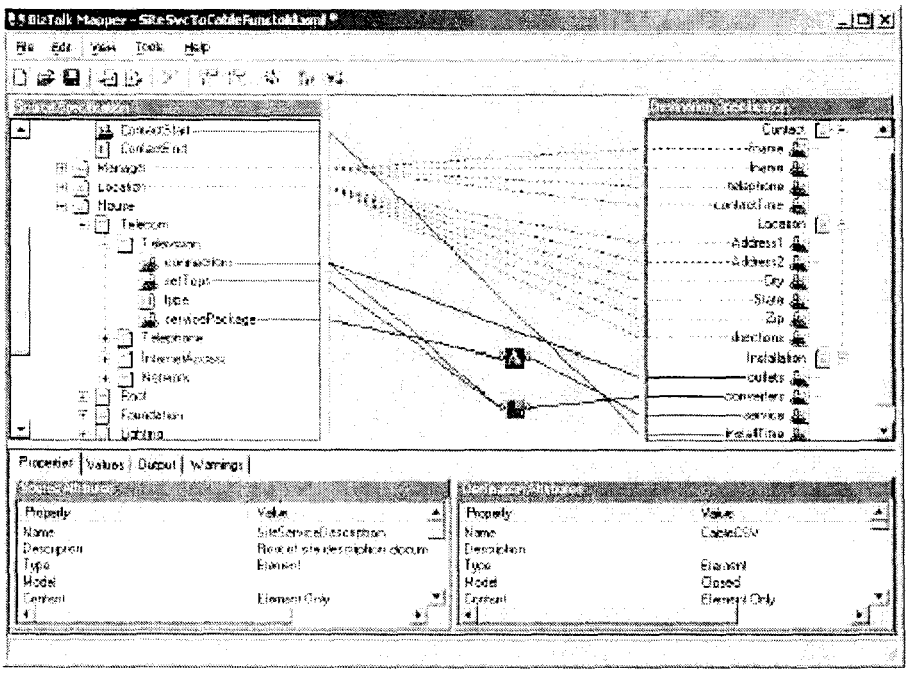


figure 24. BizTalk Mapper l'éditeur de règles de transformation de BizTalk Server

Au début de nos travaux sur les liens d'adaptation, nous avions en tête de définir un formalisme où tout type d'éléments (concept, caractéristique, type de base, ...) pouvait être lié avec tout autre type d'éléments. Si cette liberté est séduisante, d'un point de vue conceptuel elle n'est pas toujours cohérente (quel intérêt à lier le nom d'un attribut avec une classe ?). Cette liberté peut aussi déstabiliser le concepteur (c'est-à-dire ici la personne qui est chargée de définir les liens entre deux méta-modèles spécifiques). À l'inverse, des restrictions guident le concepteur sur la façon dont définir les liens d'adaptation. Ce besoin de réduire les types de liaison s'est vérifié avec l'apparition du Common Warehouse Model (CWM – version 1.0 en février 2001). Depuis une vingtaine d'années, un grand nombre de travaux ont été effectués sur la fédération de bases de données ou sur les bases de données multiples. C'est dans ses travaux que s'inscrit le data warehouse. Si au départ, "l'union" de bases de données se faisait à partir de descriptions textuelles, les solutions commerciales utilisent de plus en plus des éditeurs de liaisons (comme l'ont fait *ensuite* les solutions d'EAI). Le CWM est un ensemble de spécifications défini par l'OMG qui visent à faciliter les échanges entre entrepôts de données. Dans ces spécifications, est défini le paquetage UML *Transformation*, qui reprend les transformations usuelles sur les modèles de données en *data warehousing*. Les participants du CWM sont des acteurs importants dans ce domaine (IBM, Oracle, Unisys, Hyperion, ...). De par ce statut (importance des acteurs, expérience accumulée) nous avons réduit notre ensemble de transformations possibles et au final fortement inspiré de ces spécifications. La figure 25 représente la structure des règles de transformation du CWM.

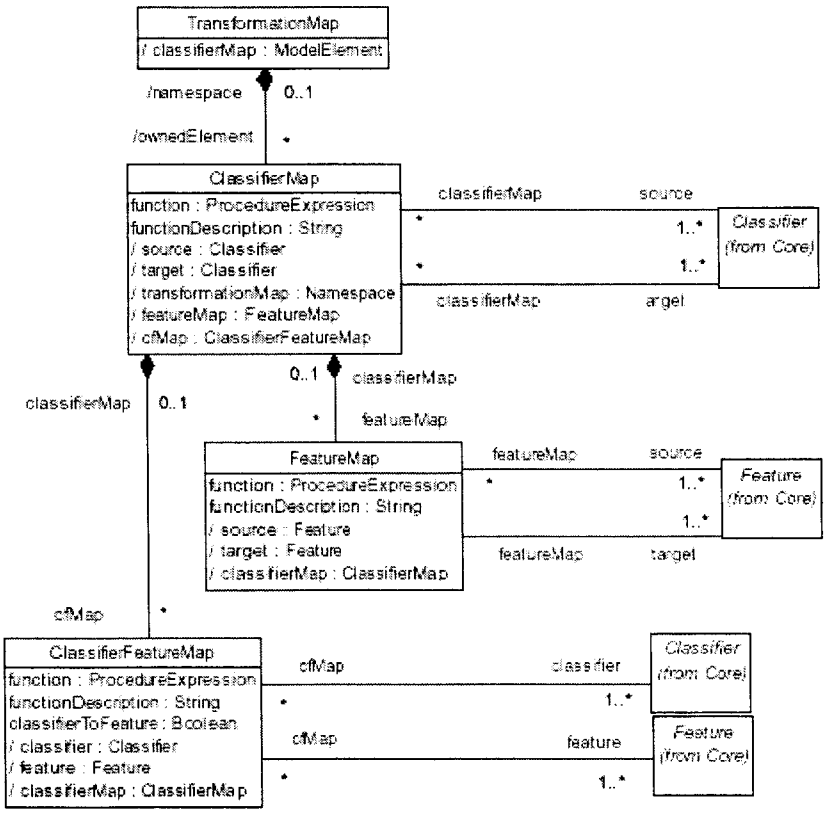


figure 25. Structure des règles de transformation du CWM

Une transformation (*TransformationMap*) est constituée d'une liaison (*ClassifierMap*) entre une source et une cible. Ces deux éléments sont des *Classifier* (des éléments qui classent d'autres éléments, ex : *Class*, *Package*, *DataType*). Cette liaison correspond à notre appellation liaison conceptuelle. Elle est elle-même constituée d'une série de lien de transformation entre les caractéristiques des deux classifiers (*FeatureMap*). Par exemple, le nom du classifier A (source) est en relation avec le nom du classifier B (cible). Ce type de liaison correspond à nos liaisons structurelles. La liaison de type *ClassifierMapFeature* qui permet de lier un classifier (contenu dans le classifier source) à une caractéristique de la cible n'a pas été intégré, pour l'instant, dans notre formalisme. Nous jugeons cet élément secondaire (les classes incluses ont rarement été utilisées dans les exemples de méta-modèles que nous avons vu).

À partir de ces deux sources d'inspiration nous avons construit un formalisme qui prend en compte la structure à trois niveaux (répercussion entre M1 et Mzéro) et le contexte d'adaptation (blocage d'accès à certaines caractéristiques, sens de lecture, différents types d'accès).

2.3 Les liaisons conceptuelles

Les liens entre concepts de type ou entre concepts d'instance sont appelés liaisons conceptuelles. Pour une plus grande précision, les expressions *liaisons de type* (CT) et *liaisons d'instance* (CI) peuvent être employées. Les liaisons conceptuelles sont le point de départ de la coopération. Lorsqu'un système se connecte, par le biais d'un AS, à un référentiel de modèles d'un autre système et demande les éléments de type A, ce sont l'ensemble des liaisons conceptuelles concernant le type A qui permettent de savoir quels sont les éléments à renvoyer et comment les adapter.

Les liaisons conceptuelles peuvent varier selon quatre facteurs :

- la cardinalité,
- la direction,
- la condition,
- le niveau du concept (liaison de type ou d'instance).

2.3.1 Liaison simple

Une liaison entre deux concepts sans autre indication est dite simple. Par exemple, un outil dans DARE correspond à une application dans COW (cf. figure 26). Le concept *Tool* est projeté vers le concept d'*Application* (i.e. Application[Tool]). Cela est toujours vrai. Le trait double pour indiquer graphiquement une liaison de projection.

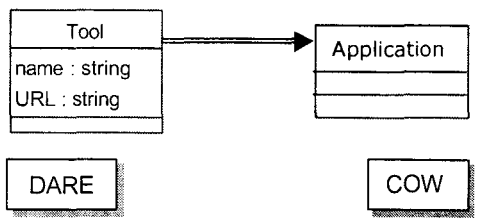


figure 26. Une liaison simple (lien de projection 1→1)

Cette liaison peut se lire de deux manières :

1. Une application 'adaptée' peut se calculer à partir d'un outil.
2. Un outil peut être projeté en une application.

L'usage de l'hypothétique ("potentialité") indique simplement qu'il peut y avoir d'autres liens conceptuels concernant *Tool* et *Application*.

2.3.2 La direction : simple projection ou équivalence

Une liaison d'adaptation a une direction indiquée par une "pointe", comme sur l'exemple précédent Application[Tool]. Elle peut avoir deux directions. Dans ce cas, il s'agit d'une liaison d'*équivalence* (en opposition avec *projection*). Il ne s'agit pas que d'un simple raccourci signifiant deux flèches en sens opposé. Il indique qu'il n'y a pas d'autres projections possibles (vers d'autres concepts). Le lien entre *Tool* et *Application* est un lien d'équivalence : un outil correspond à une application (et à rien d'autres) et une application correspond à un outil (idem).

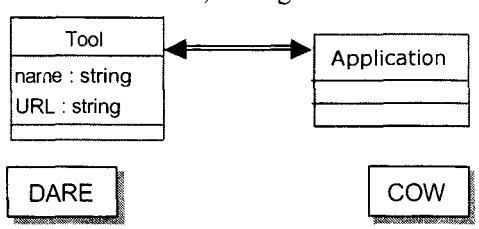


figure 27. Lien d'équivalence (1↔1)

2.3.3 La cardinalité

Un concept peut correspondre à plusieurs concepts. Ceci est très bien illustré par les relations Process[Task] et Activity[Task] (cf. figure 28). Une tâche peut être traduite en un type de processus mais aussi simultanément en un type d'activité (pour faire apparaître les sous-tâches éventuelles).

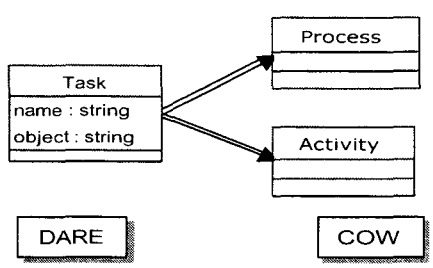


figure 28. Liaison 1↔n

La figure 28 n'indique pas les projections d'origine *Process* et *Activity* ciblées sur *Task*. Il y a un choix à effectuer pour le cas des activités composites (i.e. sous-processus) :

1. Soit un type d'activité composite donne une tâche et les types d'activités qu'il contient sont déduits de sa référence *model* sur un type de processus. On peut décider que seuls les types d'activités sont adaptés (les types de processus ne le sont pas).
2. Soit les types de processus et les types d'activité sont tous les deux traduits en tâches, et la référence *model* est traduite en relation de sous-tâche (*sub*).

Si la première possibilité paraît plus cohérente, la deuxième est pourtant préféré (cf figure 29). En effet, à l'exécution la première possibilité retire des informations importantes pour les spécialistes. Il y a des risques de pertes d'information. Prenons l'exemple de plusieurs types d'activités composites A1, A2, A3 qui référencent, à travers *model* le type de processus P. Ils sont donc composés des mêmes types d'activités, listés dans P. A1, A2 et A3 sont respectivement traduits en tâches A1', A2' et A3'. Un utilisateur de DARE qui modifie la liste des sous-tâches de A1' ne sait pas qu'il modifie la liste de A2' et A3'. Cet aspect de non-conscience est très gênant et est à l'origine de notre choix pour la deuxième solution/possibilité qui indique l'existence à part entière de la liste mais est malheureusement plus lourde à l'utilisation. Si les types de processus n'étaient associables qu'à un seul type d'activité la première possibilité eut été sélectionnée.

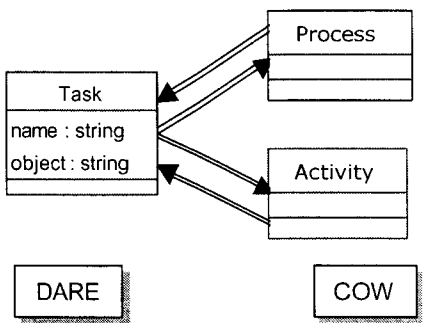


figure 29. Ensemble complet des liens entre Task, Process et Activity

Des concepts peuvent ne pas avoir de correspondances (1→0). *Operation* et *Action* de COW en sont des exemples. Cela indique que ces concepts sont inutiles pour COW.

2.3.4 Liaison Conditionnelle

Un concept peut correspondre à une entité seulement dans certaines situations. Dans l'exemple précédent de *Task* et *Process*, le choix a été fait de traduire automatiquement une tâche en un type de processus et un type d'activités. Ceci peut amener des problèmes de fonctionnement si, pour COW, la présence d'un *Process* sur la référence *model* implique obligatoirement des sous-processus. Si la traduction entraîne un type de processus "vide" (i.e. sans liste de types d'activités) Il peut très bien y avoir un dysfonctionnement. Pour éviter cette situation, il faut conditionner la projection. Sur le lien de projection peut s'apposer une condition (cf. figure 30) exprimée dans un rectangle. Elle peut être écrite sous forme d'égalités à respecter, de contraintes OCL, de fonctions booléennes, ... Le formalisme d'expression des conditions dépend de l'application de CAST sur un support particulier.

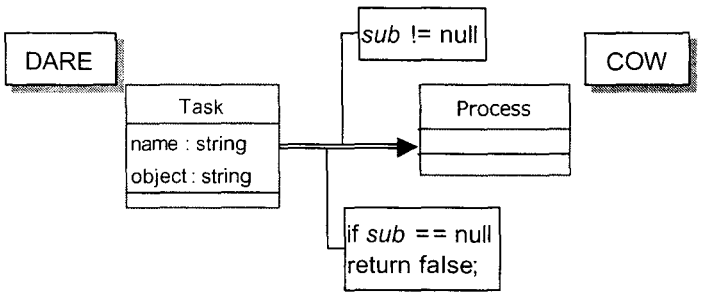


figure 30. Projection conditionnée

2.3.5 Les types de bases et les fonctions de conversion

Les types de bases (entier, flottant, booléen, chaîne de caractère, ...) sont utilisés par les concepts d'un méta-modèle à travers leurs caractéristiques. Des fonctions de conversion sont implicites à CAST. Par exemple, une liaison structurelle qui unit une propriété de type entier et une propriété chaîne de caractères fait automatiquement la conversion *entier→chaîne* : un entier 3 est traduit en "3". Néanmoins, CAST permet au concepteur de définir ses propres fonctions de conversion. Sur l'exemple précédent, le concepteur peut souhaiter "+ 3". La fonction est indiquée dans un parallélogramme (cf. figure 31).

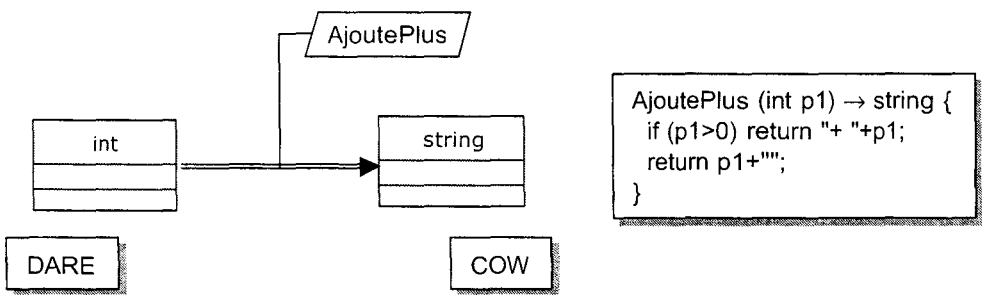


figure 31. Une fonction de conversion

Les fonctions de conversion peuvent aussi être employées quand les représentations des flottants sont différentes.

2.3.6 Le niveau du concept

Les CT et CI se lient de la même manière[39]. Néanmoins les liens entre CT ont une répercussion sur les CI. Si un CT A est projeté vers un CT B, alors le CI associé à A est projeté vers le CI associé à B. C'est une propriété intrinsèque aux concepts de type. Une liaison entre CI issue de CT n'est pas à indiquer explicitement. Elle peut apparaître à titre informatif. La figure 32 illustre la répercussion des liens entre CT sur le CI : le fait qu'une tâche A se traduise en type de processus A' implique qu'une activité de type A se traduisent en un *WfInstance* de type A'.

[39] En général, les concepts de types ne sont pas projetés vers des concepts d'instances (ni l'inverse). Même si une exception reste toujours possible (qui conceptuellement nous échapperez), nous ne fournissons aucun support pour exprimer une telle possibilité.

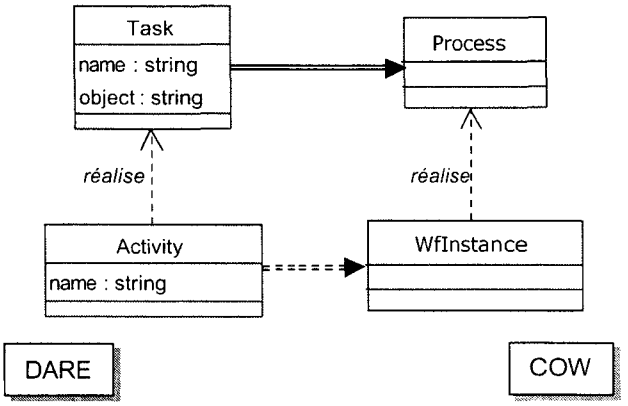


figure 32. Répercussions des liens entre CT sur les CI.

L'annotation TO (Type Only) permet toutefois de définir des liaisons entre CT n'ayant pas de répercussion sur leurs CI. Ceci s'illustre sur le lien entre *MicroRole* et *Participant*. L'objectif de cette liaison est de faire apparaître dans COW les micro-rôles en tant que participants virtuels juste à titre indicatif : les spécialistes de COW/LECS sont au courant des droits d'accès aux outils selon le type de participant. Le moyen est de transformer un micro-rôle en un participant dont le nom sera composé du nom du rôle associé + un trait bas + le nom de l'outil associé + un suffixe "*_rights*". L'attribut *description* d'un tel type de participant pourra contenir la liste des opérations autorisées. Mais il faut faire attention à ce qu'un tel type de participant ne s'instancie pas, c'est-à-dire qu'un utilisateur deviennent un tel participant. Ce lien n'a qu'un but d'aide à la modélisation. Pour cela, *Participant[MicroRole]* est annoté TO.

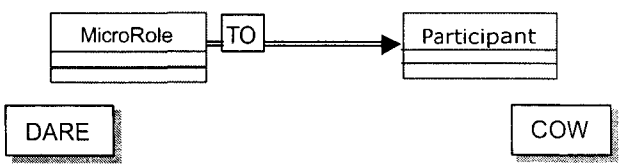


figure 33. Une liaison entre CT sans répercussion sur les CI

2.4 Liaisons structurelles

Après avoir indiqué pour chaque entité, le concept d'adaptation correspondant dans le méta-modèle 'opposé', il faut détailler la projection : comment est calculée chacune des caractéristiques de l'adaptateur. Les liaisons structurelles interviennent à cette étape. Chaque liaison est techniquement équivalente à une redirection de requête (sur une caractéristique). Cette liaison est un artefact graphique qui accélère l'implémentation. Ce qui ne peut être indiqué graphiquement peut l'être avec une fonction d'adaptation associée. C'est une possibilité propre aux liaisons structurelles. Pour le reste la liaison structurelle est simila ire à la liaison conceptuelle : elle est caractérisée par une direction, une cardinalité, une condition et le niveau des concepts qu'elle relie.

2.4.1 Une liaison simple de caractéristique

Une liaison structurelle part d'une caractéristique C1 (attribut, référence, opération, autres) et arrive sur une autre caractéristique C2. Elle peut se lire de deux manières (comme pour les liaisons conceptuelles) :

1. C1 est projetée vers C2,
2. C2 se calcule à partir de C1.

Elle est identifiée graphiquement par une flèche simple qui part d'un petit rond (qui désigne une caractéristique) et arrive sur un autre rond. Les caractéristiques héritées peuvent être affichées. Deux concepts héritant d'une même caractéristique peuvent la traiter différemment (redéfinition) et peuvent donc disposer de liaisons structurelles pour la même caractéristique. La figure 34 montre la liaison entre les noms de Tâche et d'Activity: le nom d'une activité se calcule à partir du nom de la tâche correspondante (i.e. adaptée). Il est obligatoire de faire apparaître la caractéristique name d'*Activity* héritée de *BaseClass* : il n'y a pas de concept père équivalent à *BaseClass* dans DARE.

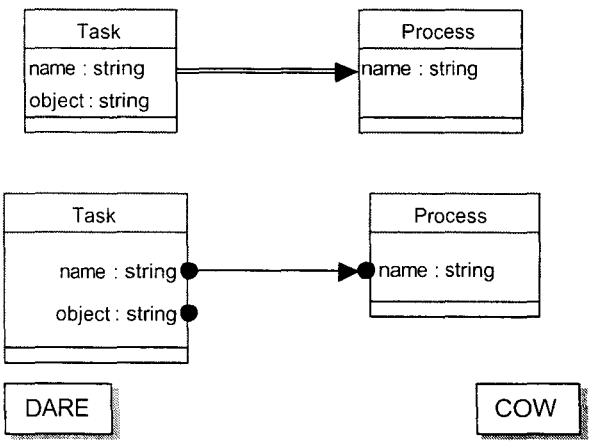


figure 34. Une liaison structurelle simple

Cette liaison peut être qualifiée de "très" simple car le type de la valeur (de la caractéristique) de départ est identique à celui d'arrivée. Dans le cas de types différents, la traduction ou l'adaptation suivra les liaisons conceptuelles. Quand la traduction nécessite des traitements à effectuer il faut alors utiliser une fonction d'adaptation ou une fonction de traduction. La figure 35 montre les autres possibilités pour exprimer l'appartenance d'un ou de liens structurels vis-à-vis de leur lien conceptuel de départ : le lien conceptuel peut-être indiqué au même endroit que les liens structurelles (à gauche) ou s'il n'y a pas d'autre association conceptuelle il n'est pas obligatoire (à droite).

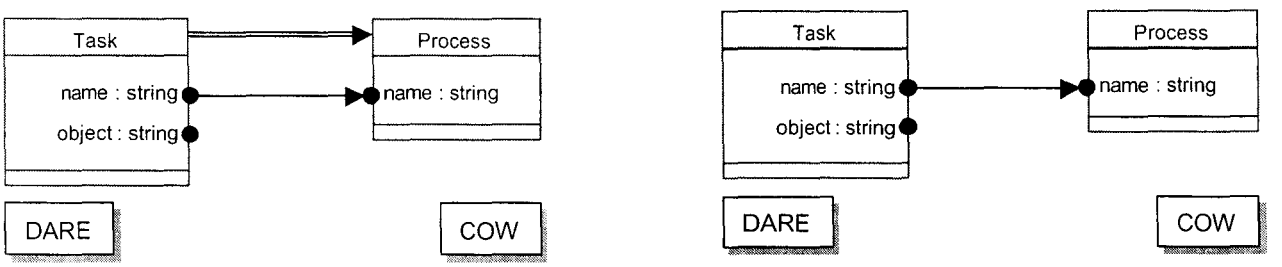


figure 35. Autres représentations de l'association lien structurel-conceptuel

2.4.2 Les fonctions de traduction et d'adaptation

Les fonctions de traduction et d'adaptation interviennent quand la notation graphique devient insuffisante pour exprimer l'adaptation ou l'utilisation de fonctions de conversion. Les fonctions de traduction sont associées aux types de bases, et les fonctions d'adaptation aux concepts. Encore une fois, le nom est choisi arbitrairement.

Fonctions de traduction. Sur l'exemple d'un adaptateur Task(Process) un traitement pour le calcul du nom peut être ajouté : comme un *Process* et une *Activity* sont susceptibles d'être adaptés en une tâche, un type de process et un type d'activité de même nom pose un problème. D'où l'adjonction d'un traitement. Pour Task(Process) le nom sera suffixé par *_wp*. Comme *WorkItem* a comme seul

équivalent *Task*, il pose aussi un problème : pour Task(WorkItem) le nom sera suffixé par *_item*. Deux fonctions de traduction apparaissent dans les règles d'adaptation (cf. figure 36).

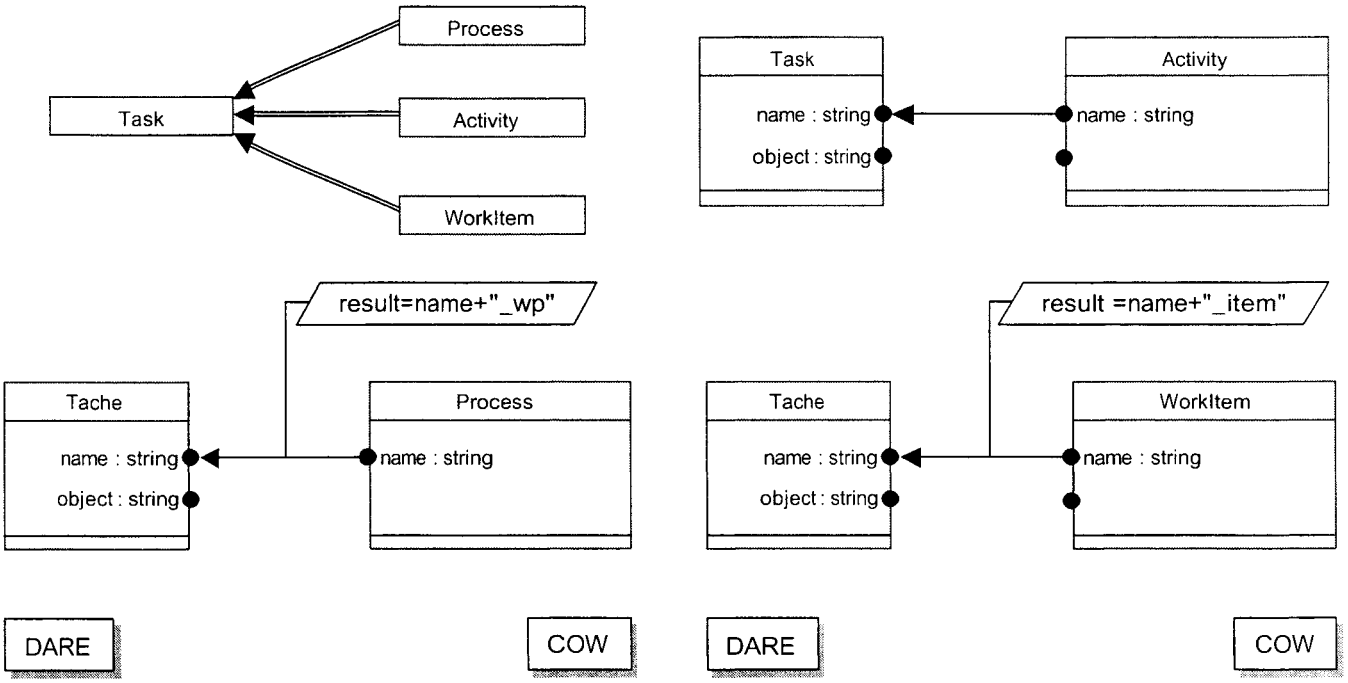


figure 36. Deux fonctions de traduction

Fonctions d'adaptation. La fonction d'adaptation s'illustre sur l'exemple déjà présenté en 2.1 - la consultation des outils d'une tâche de type *Task(Activity)*. Les outils d'une tâche de ce type sont "calculés" à partir des ressources la source (l'activité adaptée). Ce calcul nécessite un traitement (cf. figure 37 : `performers2uses`) : les ressources ne sont pas toutes des applications, certaines sont des participants. Il faut donc les trier pour ne prendre que les applications. Chacune d'elles est ajoutée au résultat après avoir été adaptée avec la fonction `adapt`.

La fonction *adapt*. Elle sert à créer un adaptateur selon les liens d'entités. C'est la fonction qui invoquée lorsqu'il n'y a pas de fonction d'adaptation indiquée. Sur l'exemple de `performers2uses`, la fonction `adapt` est invoquée de cette manière :

```
adapt (DARE,performers[i])
```

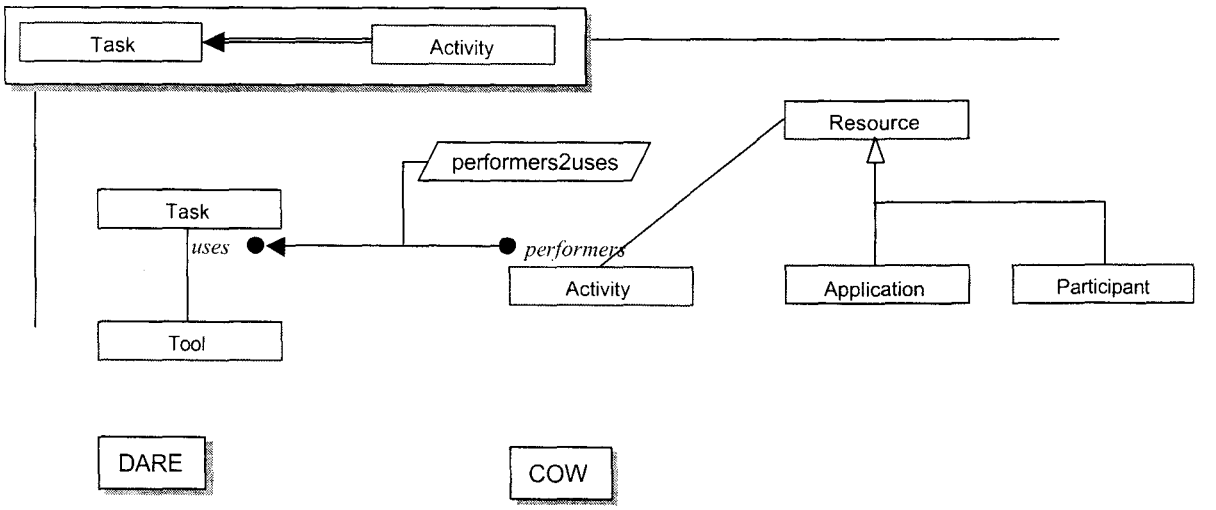
Elle cherche alors quel est le type correspondant à `performers[i]`. Comme `performers[i]` est, au moment de l'invocation, une application et que l'entité *Application* n'est liée qu'à l'entité *Tool* alors `performers[i]` est adapté en un *Tool(Application)*. S'il y a plusieurs entités d'adaptation possible il faut indiquer quelle entité choisir :

```
adapt (DARE, performers[i], Tool)
```

Sinon la fonction `adapt` prend la première liaison valide[40], c'est-à-dire celle pour laquelle `performers[i]` vérifie les éventuelles conditions.

La fonction `adapt` peut aussi prendre une liste d'éléments. Dans ce cas, elle adapte au cas par cas si aucune entité d'adaptation n'est spécifiée sinon elle adapte selon l'entité spécifiée.

[40] première selon le stockage interne !

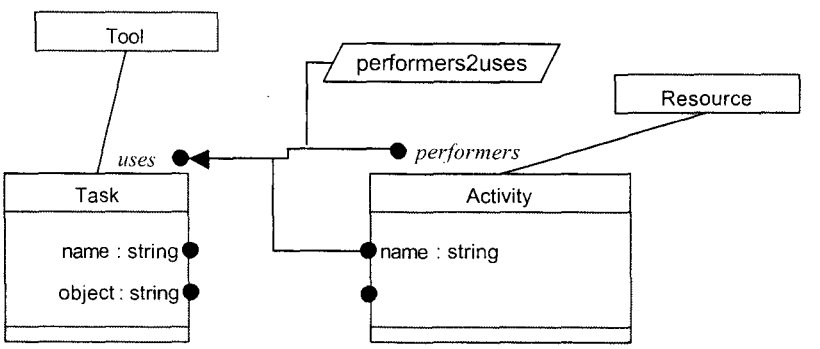



```
performers2uses
(Resource[] uses, Activity source) → Tool []
{
  for (int i=0;i<uses.length;i++)
    if (uses[i].isA(Application))
      result.add(adapt(DARE,uses[i]));
}
```

figure 37. *performers2uses* : une fonction d'adaptation

La fonction *traduct.* La fonction `traduct` est similaire à la fonction `adapt`. Elle se réfère aux fonctions de conversion et aux liens entre types de bases.

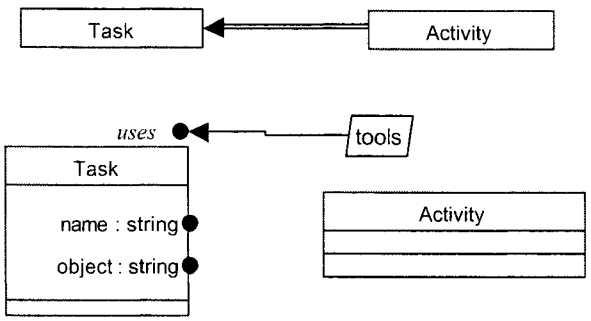
Les paramètres des fonctions de traduction ou d'adaptation sont indiqués par l'origine de la flèche. La source est aussi passée en paramètre. Donc pour avoir plusieurs paramètres utiles dans la fonction de traduction/adaptation, il faut spécifier plusieurs points de départ (cf. figure 38).




```
performers2uses
(Resource[] uses, string name, Activity source) → Tool []
{
  for (int i=0;i<uses.length;i++)
    if (uses[i].isA(Application))
      result.add(adapt(DARE,uses[i]));
}
```

figure 38. Fonction à plusieurs caractéristiques de départ

Logiquement, une fonction de traduction ou d'adaptation n'a pas obligatoirement de points de départ à avoir (cf. figure 39).




```
tools
(Activity source) → Tool []
{
  Resource[] resources=source.uses();
  for (int i=0;i<resources.length;i++)
    if (resources[i].isA(Application))
      result.add(adapt(DARE,resources[i]));
}
```

figure 39. Fonction sans point de départ

La source étant toujours indiquée, il suffit d'accéder aux caractéristiques de la source. Néanmoins, les points de départs servent :

- à bien expliciter la projection. Des liaisons graphiques sont plus significatives que de simples paramètres dans une fonction.
- Lorsqu'il n'y a pas de traitement à effectuer, la liaison correspond alors à la fonction de traduction ou d'adaptation. Il est plus rapide d'indiquer un simple lien qu'une fonction.

Si la valeur d'une caractéristique est issue de la fusion de la valeur de deux caractéristiques, et qu'il s'agit d'une simple fusion (sans traitement pour l'ordonnancement), alors deux points de départs suffiront. Le but de CAST étant d'accélérer au plus possible l'implémentation, la fusion est une fonctionnalité intégrée à celui-ci[41] (cf. figure 40).

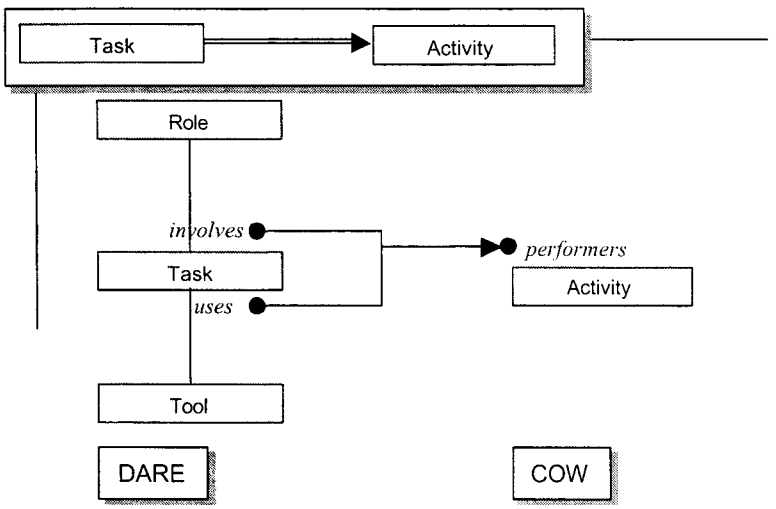


figure 40. Fusion de caractéristiques

La sélection par type, telle qu'elle existe dans `performers2uses`, est aussi une fonction, à notre avis, fréquente qui doit être intégrée.

[41] et doit donc faire partie de l'application de CAST à un support

2.4.3 Mécanismes d'extension

Par analogie avec certains langages de programmation qui disposent de mécanismes d'intercession et proposent la modification du look-up, CAST offre la possibilité de modifier les mécanismes d'adaptation qui régissent le moteur "d'inférence" des liaisons à l'exécution d'un service d'adaptation. Ces mécanismes ont des dispositifs d'extension par lesquels peuvent s'ajouter par exemple la fusion et la sélection de la section précédente. Deux fonctions/mécanismes principales sont redéfinissables : `conceptLinkApplication` et `featureLinkApplication` respectivement les fonctions d'application d'un lien d'adaptation conceptuel et structurel. L'ajout de nouvelles fonctionnalités comme la fusion ou la sélection peuvent s'effectuer en intégrant des redirections dans `featureLinkApplication`.

```
featureLinkInvocation ( FeatureLink featureLink,
                        Argument[] arguments) {
 if (featureLink.parameters.lengh>1
   && featureLink.function==null)
  fusion(featureLink, arguments);
 if (featureLink.parameters.lengh==1
    && featureLink.returnType.isAConcept()
    && correspondantConcepts(featureLink.parameters[0])==null
    && featureLink.parameters[0].isFatherOf(
correspondantConcepts(featureLink.returnType)) )
    selection(featureLink, arguments);
  ...
  else
    adapt(featureLink,arguments);
}
```

La liste des types et des fonctions associées aux mécanismes d'extension est présentée dans l'annexe C.

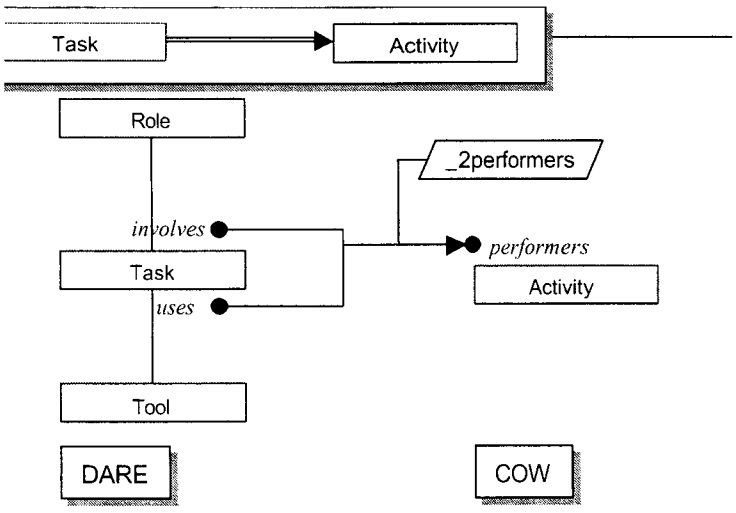
2.4.4 Les différents types d'accès aux caractéristiques

Une caractéristique peut être consultée, éditée de plusieurs façons :

- Lecture, affectation pour les caractéristiques à valeur unique.
- Ajouter, modifier, suppression, lecture d'un élément.

Il faut parfois indiquer pour chacun de ces accès, un traitement particulier. Par exemple la lecture des outils est "différente" consiste en un traitement différent de l'ajout d'un seul outil. Les correspondances directes une liste-une liste ou un élément-un élément n'ont pas à justifier un traitement pour chaque type d'accès. Les précédentes fonctionnalités *fusion* et *selection* prennent en compte chaque type d'accès.

Pour indiquer un traitement pour un type d'accès particulier, il faut utiliser un des préfixes suivants devant le nom de la fonction indiquée dans le parallélogramme : set_, add_, modify_, remove_, getAt_. L'absence d'un préfixe correspond à la lecture simple. La figure X montre la définition de traitements pour deux types d'accès sur l'exemple `_2performers` (dans le cas où la fusion ne serait pas une fonctionnalité intégrée à CAST).




```
_2performers

add__2performers
(Resource newValue, Task target,
 Activity source) → void
{
  if (newValue.isA(Application))
    target.uses.add(
      adapt(DARE,newValue,Tool) );
  else
    target.involves.add(
      adapt(DARE,newValue,Role) );
}

modify__2performers
(Resource newValue, Resource oldValue,
 Task target,  Activity source) → void
{
  if (oldValue.isA(newValue.type)) {
    if (oldValue.isA(Application))
      target.uses.modify(
        adapt(DARE,newValue,Tool)
        adapt(DARE,oldValue,Tool) );
    else
      target.involves.modify(
        adapt(DARE,newValue,Role)
        adapt(DARE,oldValue,Role) );
  }
  else {  ...   }
}
```

figure 41. Les fonctions d'ajout et de modification de *performers*.

Le "sens" de lecture est encore très important. Sur l'exemple, la fonction `add__performers` s'occupe d'indiquer comment ajouter un performer dans un type d'activité adaptée selon Activity(Task). Si la fonction de lecture n'est pas spécifiée alors c'est la fonction de base associée (la fusion) qui est employée.

Pour **indiquer les accès non autorisés**, n'associer aucun lien structurel à une caractéristique suffirait : la lecture de celle-ci sur un adaptateur renverrait *null* Comme pour les caractéristiques n'ayant aucune correspondance. Néanmoins, d'un point de vue sémantique, il est préférable d'indiquer explicitement que le système coopérant cache certaines informations. Pour certains systèmes, il serait par exemple intéressant de renvoyer une exception. La caractéristique *model* sur la figure 42 est un exemple d'accès non autorisé : il s'agit d'une coopération où HLC ne montre que des tâches de haut niveau sans préciser leur contenu (cela peut être une politique adoptée dans un contexte très compétitif [CAN 01])

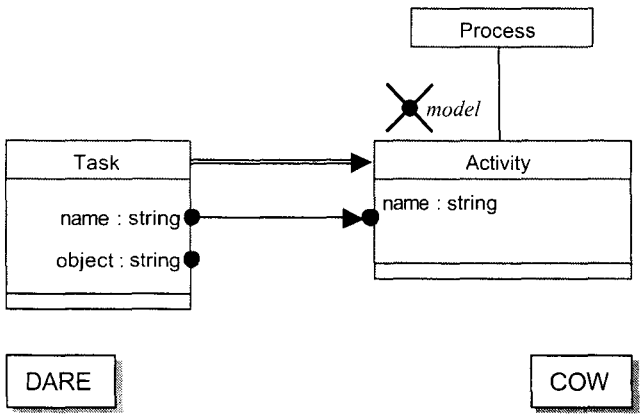


figure 42. La référence *model* non accessible

2.4.5 Limitation à deux systèmes/méta-modèles

Le formalisme présenté est un support à la définition de liens d'adaptation. Un ensemble de liens permet la génération d'un service d'adaptation spécifique à des méta-modèles. Pour l'instant CAST n'autorise que la liaison entre deux méta-modèles c'est-à-dire deux systèmes[42] : si HLC souhaiterait coopérer avec une autre société (dont le système serait ni COW ni DARE) il lui faudrait définir (séparément) un nouvel ensemble de liens d'adaptation entre DARE et le système qu'utilise ce nouveau partenaire et générer ensuite un autre AS. Les liens d'adaptation ne peuvent s'effectuer entre trois méta-modèles (ou plus), et par conséquence un AS ne peut gérer la coopération entre trois systèmes différents. C'est une limite provisoire que nous espérons supprimer.

L'inconvénient majeure est l'augmentation du nombre d'AS : pour trois systèmes différents, on peut avoir besoin de trois AS différents (s'il faut une coopération entre chacun d'eux), pour quatre systèmes c'est sept AS différents potentiellement nécessaires... Deux remarques doivent être faites vis-à-vis de cet inconvénient :

- la création de l'AS est rapide (notre objectif),
- le contexte commercial/économique implique rarement une coopération "complète" (c'est-à-dire entre chaque système présent dans la chaîne de valeurs).

Malgré l'inconvénient de l'augmentation rapide du nombre d'AS, CAST reste viable actuellement. La possibilité de créer son propre langage pivot ou d'effectuer des liens concernant trois systèmes/méta-modèles ou plus est évoqué dans le chapitre 7.

Une fois les liens d'adaptation créés, l'AS est généré. Le moteur de gestion des adaptateurs et ces derniers ont une structure particulière appropriée à notre contexte de coopération. La démarche qui nous a amenés à définir ces structures est l'objet de la partie suivante.

3 Fonctionnement

3.1 Rappel du fonctionnement d'un adaptateur

La notion d'adaptateur est un concept assez répandu en informatique. Le principe du patron de conception (*design pattern*) *Adapter* désormais célèbre et défini dans [GoF 95] est qu'un objet adaptateur fournit les fonctionnalités promises par une interface sans indiquer quelle classe

[42] Nous aborderons les problèmes d'accès informatique dans le chapitre 5.

implémente celle-ci. Par exemple, un ensemble d'objets est implémenté de manière à accéder à des fonctionnalités définies dans une interface I1. D'autres fonctionnalités de même type existent mais leurs points d'entrés sont définis différemment dans une interface I2. La classe adaptatrice implémente l'interface I1 en invoquant les fonctionnalités définies par I2. Grâce à cette classe, les précédents objets peuvent accéder à ces autres fonctionnalités similaires.

Il y a deux possibilités pour construire un adaptateur. Sur l'exemple d'une objet de type adapté un objet de type B :

- Surcouche : un type B' est créé. Il spécialise B et contient un objet de type A. Chaque requête vers B' est transformé en requête A pour l'objet contenu, et l'inverse pour le résultat. (cf. fig 43,à gauche).
- Fusion : un type B' est créé. Il hérite de A et B. Les requêtes de type B sont transformé en requêtes de type A. Il n'y a pas d'objet contenu cette fois. (cf. fig 43, à droite).

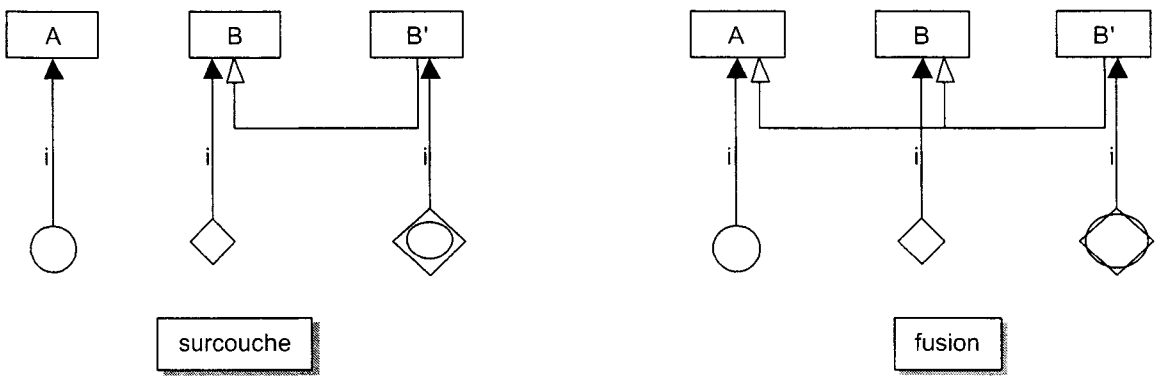


figure 43. Deux méthodes d'adaptations

3.2 Choix du type d'adaptateur

Il nous faut des adaptateurs bi-directionnels (*two-way adapter*). C'est-à-dire qu'un objet peut être manipulé dans son contexte d'origine et dans un contexte externe en étant adapté. En général, la pratique du *surcouchage* n'est pas considérée comme bi-directionnelle. L'objet adapté n'est utilisé qu'à travers l'adaptateur, il n'a pas de vie propre. Il s'agit surtout d'adaptation de la classe plutôt que des objets. Par contre, la pratique de la fusion (avec l'héritage multiple) permet à l'objet d'être manipulé dans les deux contextes. Cela a été notre premier choix [LEP 01]. Mais à travers ce premier travail, nous avons discerné quatre problèmes vis-à-vis de la fusion :

Une modification inévitable du référentiel. Un système utilise un référentiel défini avec un ensemble de types[43] équivalents aux concepts du méta-modèle utilisé. Des adaptateurs basés sur la fusion n'impliquent pas une surcouche : les éléments du référentiel sont donc hybrides. Lorsqu'un système A souhaite coopérer avec un système B (méta-modèles différents) il doit modifier le type des objets pour qu'ils soient hybrides ! La même opération doit s'effectuer pour le système B. Le degré d'autonomie chute considérablement.

À chaque nouvelle coopération, il faut modifier l'implémentation du référentiel. En plus d'être très exigeant, cela peut donner des types héritant d'une dizaine d'autres types. Parallèlement le système doit s'arrêter à chaque nouvelle coopération à cause du changement d'implémentation. Ceci constitue un autre point ennuyeux. L'utilisation de langages réflexifs, avec des mécanismes d'intercession, est une solution : employer les liens d'héritage, implémenter de nouvelles opérations. Malheureusement

[43] Un type est ici la projection d'un concept dans le support d'implémentation. Adoptant le paradigme objet, nous aurions pu employé le terme "classe" mais nous préférons pour l'instant être plus général.

les langages et intergiciels réflexifs (avec intercession) restent actuellement peu utilisés. Il est préférable de se baser sur des supports d'implémentation répandus.

Un problème de nommage. Deux types A et B qui doivent être adaptés/fusionnés ont deux point d'entrées (attributs, références, opérations, méthodes) de même nom et de même type. Toutefois ces points d'entrées n'ont pas la même signification. Par exemple, le nom peut être une référence unique (une sorte de nom qualifié qui lui sert d'identifiant comme une URI ou un IOR) pour l'un et un nom indicatif pour l'autre (il dispose alors d'un autre attribut d'identité). Il n'y a pas de solution possible à ce problème ![44]

Un problème de typage. Deux types A et B qui doivent être adaptés/fusionnées ont deux point d'entrées de même nom mais cette fois-ci de types différents : un point d'entrée *id* qui est un entier long pour A et une chaîne de caractères pour B. Encore une fois il n'y a pas de solution possible.

Des problèmes accentués par le 1 ↔ n possible. Un concept peut correspondre, dans une adaptation, à plusieurs concepts (*Task → Process, Activity, WorkItem*). Les problèmes de nommage et de typage sont accentués par cette caractéristique car, pour une seule coopération, les types hybrides peuvent hériter de 3, 4 ou plus de types. Le risque de conflit est encore plus grand.

Les problèmes de modification du référentiel, de nommage et de typage ne pouvant être résolu (sans avoir recours à des plate-formes spécifiques), notre choix se porte logiquement vers le surcouchage. Ce dernier ne pose pas les trois précédents problèmes. Néanmoins cette solution n'est pas considérée comme bi-directionnelle ... et elle ne l'est pas. Il nous faut donc rendre le surcouchage bidirectionnel. Il y a trois aspects à considérer pour cela : la création d'une adaptateur, l'identité et l'adaptation inutile. Dans notre utilisation, il s'agit de surcoucher les objets et non pas les "classes".

3.3 Rendre la surcouche bidirectionnelle

3.3.1 L'instanciation

Dans le patron de conception de base, la création d'un adaptateur implique la création d'une instance de la classe d'origine. Cette instance ne vit qu'à l'intérieur de l'adaptateur et n'est donc pas manipulée directement. Il s'agit du fonctionnement général. Manipuler directement l'instance revient à perdre l'unité que forme l'adaptateur et la source. Appliquer une certaine autonomie à l'objet source peut entraîner des incohérences. Si l'adaptateur a mis en place un filtre d'affectation pour la valeur du nom (une propriété de l'adaptateur et de l'objet source) pour n'accepter qu'un certain ensemble de caractères et que l'objet source n'a pas ce type de filtre, l'affectation d'une valeur pour le nom directement sur l'objet source peut affecter un nom interdit pour l'adaptateur. La valeur du nom serait incohérente avec le contexte de l'adaptateur. Il est permis de penser que ce type de problème (affectation de valeurs interdites) se posera dans notre utilisation de tels adaptateurs. Ce n'est pourtant pas le cas : dans notre contexte de coopération, le concepteur implémente les méthodes de l'adaptateur *en ayant à l'esprit d'adapter un objet qui a une vie propre*. Dans le cas précédent du nom, le développeur met un filtre à l'affectation mais aussi à la consultation, pour modifier les noms incohérents.

Un test d'égalité entre l'adaptateur et la source donne, avec la surcouche, un résultat négatif. L'adaptateur fusionnel évite ce problème. C'est une des raisons d'éviter à la source d'avoir une vie propre. Mais, encore une fois, ce genre de test n'a pas lieu dans notre contexte, car chaque système ne voit qu'un seul type d'objet : soit l'adaptateur pour le système hôte, soit la source pour le système d'origine.

[44] Des mécanismes d'invocation qui permettraient à un point d'entré de varier sa réponse selon l'invocateur est une réponse. Dans un contexte distribué (notre cas), il est difficile de connaître tous les types d'invocateurs possibles : pour des types inconnus, quelle information renvoyer ?

L'objet adapté ayant une vie propre, la création d'un adaptateur doit être différente : il adapte un objet déjà existant. L'instanciation (d'un adaptateur) prend donc un paramètre : l'objet source. Ceci amène au problème de l'identité.

3.3.2 Identité

Si les adaptateurs sont créés pour adapter des objets existants, il faut veiller à toujours avoir, pour un objet source, le même adaptateur. Ceci permet au système utilisant les adaptateurs de faire directement des tests d'égalité. Sur notre exemple, une requête est envoyée à Process(Formation) pour connaître sa première activité. La sous-tâche *POO_Mod* est retournée en étant adaptée Activity(Task). Un adaptateur Activity(*POO_Mod*) est créé. Si Process(Formation) reçoit la même requête, il ne doit pas créer un autre adaptateur pour *POO_Mod* car un test d'égalité renverrait *faux* ce qui ne correspond pas à la réalité. La présence d'une table des adaptateurs existants est donc nécessaire. La fonction *adapt* utilise cette table.

Une autre solution est d'utiliser une méthode de test d'égalité spécifique (comme la méthode *_is_equivalent* en CORBA) qui testerait l'égalité des sources. Néanmoins les systèmes existent déjà, l'implémentation est déjà présente, et les tests d'égalité déjà codés. Il est irréaliste de contraindre les systèmes à modifier leur implémentation pour intégrer une autre opération de test d'égalité. De plus, si un système utilise un middleware qui l'oblige déjà à utiliser une opération spécifique de test d'égalité, cela obscurcirait ou complexifierait de surcroît l'implémentation des référentiels.

[élément structurel n°1] Avoir une table de correspondance : l'objet et l'adaptateur correspondant (existant ou nul).

3.3.3 Adaptation inutile

Il ne faut pas adapter inutilement des objets. Plus précisément, il ne faut pas adapter un adaptateur de type A(B) pour le système B, mais simplement renvoyer la source (plutôt qu'un objet *B(A(B))*). Un utilisateur de LECS parcourt *Formation_SP* de HLC à l'aide de *Formation_SP'* qui est de type Process(Task). S'il demande la liste des activités, *Spanish* va apparaître dans celle-ci. *Spanish* ne doit pas apparaître en tant que Process(*Spanish'*) où *Spanish'* est l'adaptateur Task(*Spanish*) utilisé par HLC. *Spanish* doit donc apparaître en tant que *Spanish* (et non pas *Process(Task(Spanish))*).

Si les adaptations inutiles ne sont pas évitées, des adaptateurs constitués d'une dizaine de couches superflus peuvent apparaître (ralentissant le système et consommant de la place mémoire inutilement). C'est la fonction *adapt* qui se charge d'éviter ce type de situation. Chaque adaptateur doit avoir une référence sur l'objet source. Celle-ci doit être la même tout le temps de manière à être utilisable par la fonction *adapt*.[45]

[élément structurel n°2] La fonction *adapt* évite les adaptations inutiles. Tous les adaptateurs ont un lien bien défini vers la source.

3.3.4 Les adaptations 1 ⇔ n

Un objet peut être adapté en différents types d'objets. Par exemple, une tâche peut être adaptée en un *Process*, une *Activity* ou un *WorkItem*. Ceci implique que la table des adaptateurs existants possède pour chaque objet source, les adaptateurs associés et le type d'adaptation correspond. Le type d'adaptation n'est pas forcément le type de l'adaptateur : il peut s'agir d'un sous-type. C'est pour cette raison que la fonction *adapt(mm, source)* se décline en *adapt (mm, source, type souhaité)*.

[élément structurel n°3] La table des adaptateurs existants est aussi indexée par le type d'adaptation choisi.

[45] Cette directive est encore plus importante, si on étend notre approche à l'utilisation de trois systèmes ou plus.

3.3.5 Droits d'accès

Tous les éléments d'un système ne sont pas accessibles. Il faut respecter la contrainte de confidentialité dans le cadre de l'interopérabilité politique (cf. chapitre 2). C'est aux utilisateurs à indiquer les éléments exportables. L'indication doit être faite pour chaque système avec lesquels ils coopèrent : certains éléments seront visibles pour certains et non visibles pour d'autres.

Lorsqu'un élément est accessible de l'extérieur, les entités externes peuvent le manipuler selon les opérations permises (lecture, ajout, suppression, affectation et/ou modification). Mais il peut être décidé que certains éléments ont une plus grande restriction d'accès. Si toutes les tâches exportées sont "totalement" manipulables, il est profitable de pouvoir indiquer que *Formation_SP_1* (un clone) n'est accessible qu'en lecture ou que certains de ses champs ne sont accessibles qu'en lecture. De cette manière, il est possible de mettre une (in)visibilité sur les éléments et sur leurs caractéristiques.

[élément structurel n°4] Il faut pouvoir mettre des contraintes d'accès sur chaque caractéristique d'un objet-source.

3.3.6 Flexibilité

Un de nos objectifs vis-à-vis des AS est de leur attribuer une grande flexibilité. Nous considérons un AS comme un modèle de liaison de méta-modèle qu'il doit être possible de modifier. Un de nos objectifs est de mettre en place un environnement de prototypage pour tester le service d'adaptation avant de l'intégrer dans les systèmes. Mais le prototypage ne permet "seulement" que d'éviter des erreurs et de vérifier la cohérence sur des exemples, au moment de la conception. L'abandon de certains choix de liaisons ou la modification de certaines fonctions peuvent n'intervenir qu'après une longue période d'utilisation. Les motifs de ces changements ne sont visibles qu'après une expérience conséquente du service d'adaptation et sont sûrement propres à un contexte d'utilisation précis. C'est pour cette raison qu'il faut associer à ce service, en plus d'un outil de prototypage, des mécanismes de modifications dynamiques.

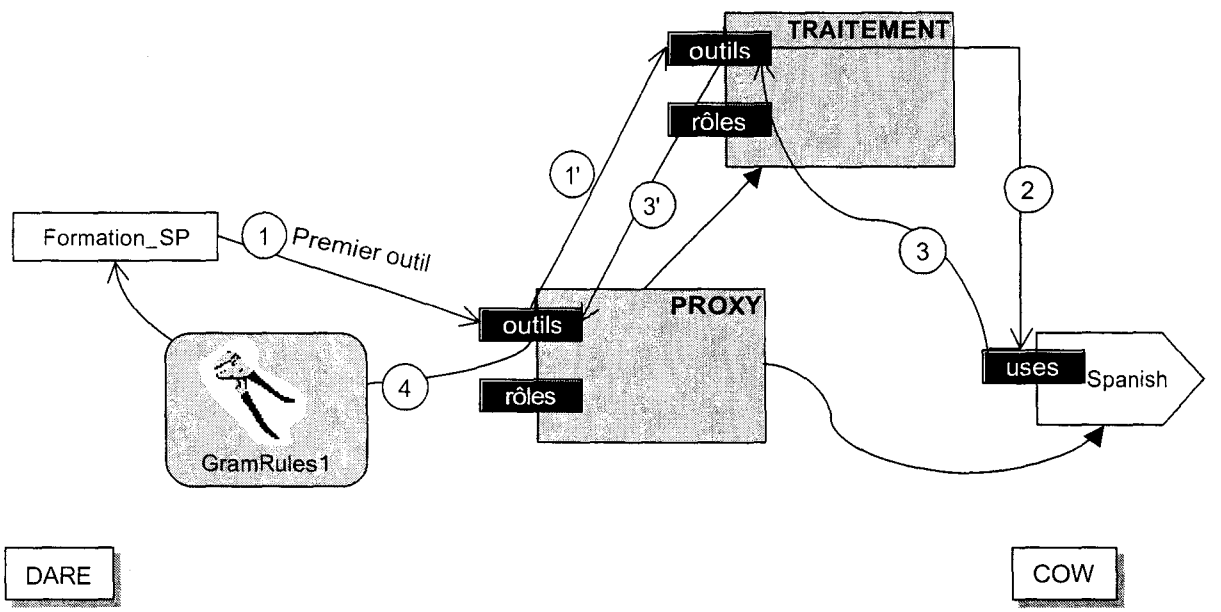


figure 44. Décomposition d'un adaptateur

Deux types d'éléments peuvent être modifiés : les liaisons d'entités et les liaisons de caractéristiques. Pour évoquer les contraintes sur la structure des adaptateurs, il est nécessaire d'indiquer que l'utilisation d'un intergiciel, pour le(s) support(s) de CAST, est inévitable pour rester dans des normes actuelles. Nous ne pouvons pas supposer, donc, d'avoir à disposition des mécanismes réflexifs puissants. C'est la structure des adaptateurs qui se doit alors d'être suffisamment générique

pour accepter les changements. Pour supporter les modifications des liaisons de caractéristiques, chaque adaptateur aura une référence sur la source mais aussi une référence sur un objet "traitement" (cf. figure 44). L'implémentation des points d'entrée de l'adaptateur consiste en une redirection vers l'objet traitement. Ce dernier a les mêmes points d'entrée que l'adaptateur. Leur implémentation correspond aux liaisons de caractéristiques. Ainsi s'il y a des modifications à effectuer, chaque adaptateur "modifié" ne change que son objet traitement. Cette méthode par "proxy" est très souvent utilisée pour apporter des mécanismes d'intercession spécifiques, comme modifier dynamiquement l'implémentation d'opérations, dans des langages de programmation [TAN 01] ou des intergiciels [RAV 99]. Un adaptateur *Task(Activity)* et un adaptateur *Task(Process)* sont identiques au niveau de l'implémentation du proxy. Seuls les objets traitement, qui leur sont affectés à leur création, sont différents.

[élément structurel n°5] La partie traitement d'adaptation d'un adaptateur n'est pas intégrée dans son implémentation.

Les cinq éléments structurels sont les guides de conception du moteur de gestion des adaptateurs et de la structure générique des adaptateurs. La conception est décrite précisément dans la partie suivante.

4 Modèle Conceptuel *

Le modèle conceptuel d'un service d'adaptation est décrit, dans cette partie, selon l'ordre suivant :

- La structure générique des adaptateurs : les objets traitements, les filtres, les proxys.
- Le moteur de gestion des adaptateurs (appelé ici le *service*) : les tables de droits d'accès, d'adaptateurs et de projection et le service (point central).
- Une vue de l'ensemble englobant et situant l'ensemble des éléments précités.
- Des diagrammes de séquences pour préciser la sémantique (fonctionnement conceptuel) de chaque élément de l'architecture.

4.1 L'adaptation

4.1.1 La classe Adapter et les paquetages X_Adapter

La classe *Adapter* (cf. figure 45) définit les caractéristiques communes de tous les adaptateurs ou selon les propos de 3.3.6, les proxys. La première caractéristique est la référence vers la source que l'adaptateur adapte à un contexte donné. La contrainte de flexibilité nous a apporté une contrainte dans la structure des adaptateurs (élément structurel n°5) : déléguer le traitement de manière à pouvoir changer celui-ci. L'adaptateur dispose d'une référence vers un objet "traitement" (*ProcessingObject*). Enfin l'adaptateur référencie aussi un filtre qui permet de respecter des contraintes d'accès à la source (élément structurel n°4). C'est à travers ce filtre que sont effectués les accès à la source par l'objet traitement.

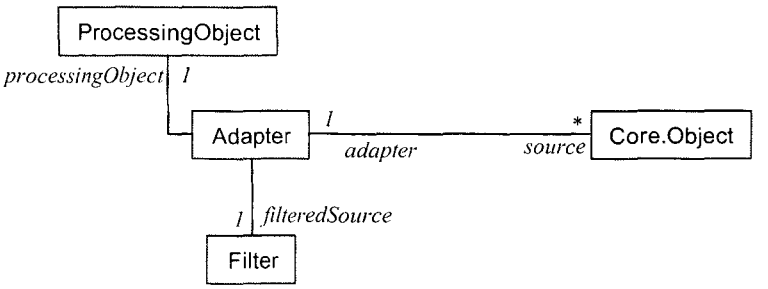


figure 45. La classe Adapter

Après avoir défini les liens de projection, CAST génère un paquetage pour chacun des méta-modèles. Le nom de ces paquetages est le même que les méta-modèles avec le suffixe *_Adapter*. Ces paquetages contiennent les classes qui permettent d'adapter des éléments extérieurs à leur méta-modèle associé.

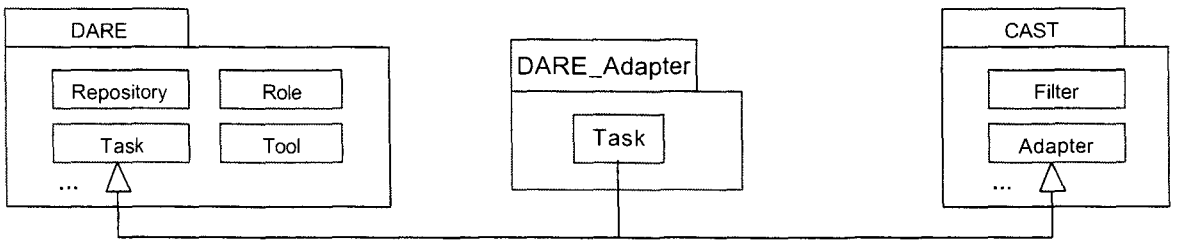


figure 46. Le paquetage DARE_Adapter et sa classe Task

Par exemple, le paquetage *DARE_Adapter* (cf. figure 46) contient des classes qui héritent des classes du paquetage *DARE*, c'est-à-dire Task, Role, Tool, ... L'objectif de *DARE_Adapter* est de fournir les classes permettant d'adapter des modèles non DARE à DARE. Par exemple la classe *DARE_Adapter.Task* est utilisée quand un type d'activité (de COW) est adaptée (en une tâche) dans le système DARE. En d'autres termes, une instance de la classe *DARE_Adapter.Task* adapte l'activité. Nous revenons plus précisément sur les classes d'adaptation ultérieurement.

4.1.2 Les filtres : la classe Filter et les classes XFilter

Les paquetages *X_Filter* sont des paquetages générés en même temps que les *X_Adapter*. Ils contiennent des classes dont l'objectif est de filtrer les accès de l'extérieur aux instances du méta-modèle associé. Ces classes héritent de la classe *Filter* (cf. figure 47). La classe *Filter* définit les caractéristiques communes de tous les filtres.

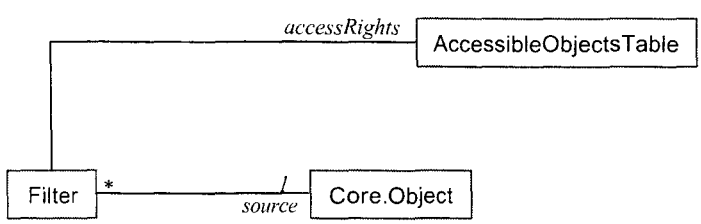


figure 47. La classe Filter

Un filtre dispose d'un lien vers la source et la table d'accès (cf. 4.2.1). Avant de renvoyer l'invocation vers la source, le filtre consulte la table des objets accessibles pour savoir si le champ

accédé est consultable/modifiable. Une classe *XFilter* hérite de *Filter* et de la classe pour laquelle elle doit filtrer les accès (de l'extérieur) aux instances.

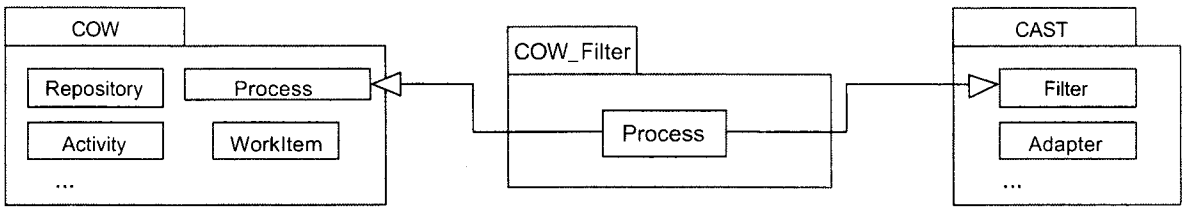
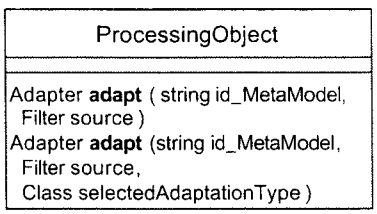


figure 48. *COW_Filter.Process* : un filtre pour les *Process*

La figure 48 montre l'exemple de *COW_Filter.Process*[46]. Cette classe, générée par CAST, hérite de *Filter* et *Process*. Chaque méthode définie par *WorkflowProcess* y est redéfinie pour constituer une redirection conditionnée par l'accessibilité du champ.

4.1.3 Les traitements : la classe ProcessingObject et le paquetage processingObjectsRepository



Le traitement d'adaptation (défini lors de la définition des liens de projection) qu'effectue un adaptateur est en fait délégué à un objet traitement. Les objets traitements sont propres à chaque entité à adapter mais dispose de caractéristiques communes définies par *ProcessingObject*. Il s'agit simplement des deux opérations *adapt* possibles évoquées en 2.4.2. Ainsi le code défini dans les fonctions de traduction et d'adaptation peut directement être placé dans l'implémentation d'une classe de traitement. Lorsqu'une invocation de l'opération *adapt* est faite dans ce code, c'est la méthode *adapt* de l'objet traitement qui est invoquée. Celle-ci est bien sûr une redirection vers la réelle opération *adapt* (contenu dans *Service*). La seule différence est que les méthodes *adapt* d'un objet traitement prennent en paramètre un objet filtré. Cela provient du fait qu'un objet traitement ne manipule que des objets filtrés.

[46] Cette classe, plutôt que DARE_Filter.Task, nous paraît plus pertinent. Elle apparaît dans un diagramme de séquence avec *DARE_Adapter.Task* dans le cadre d'un adaptateur Tache(Process).

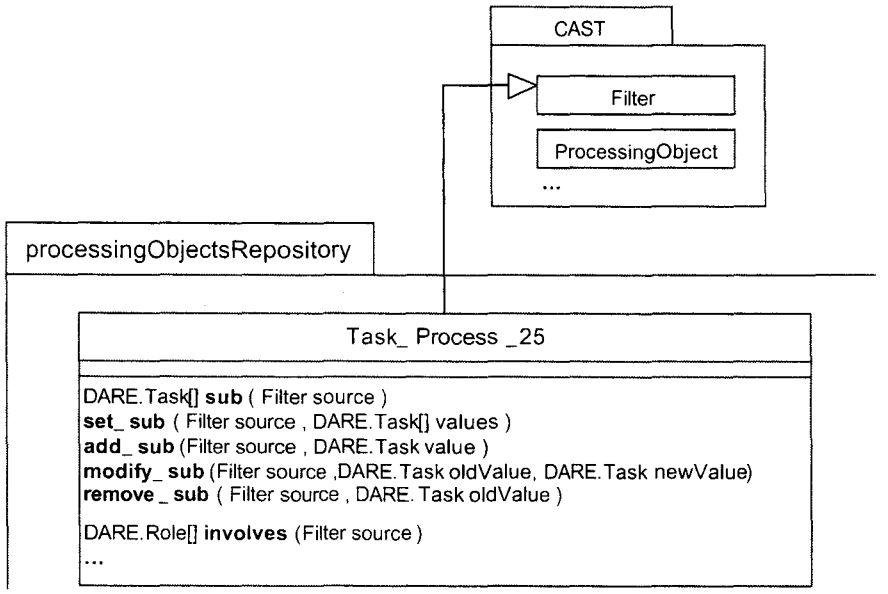


figure 49. Traitement Task (Process)

Un paquetage *processingObjectsRepository* est conçu pour recevoir toutes les classes traitements qui seront générées à la suite de la définition des liens de projection. Ce paquetage, associé au service, n'a pas de réellement signification conceptuelle mais indique le fonctionnement à l'exécution. Chaque classe qu'il contient hérite de *ProcessingObject* et reprend toutes les opérations de l'entité, pour laquelle elle fournit les traitements d'adaptation, en y ajoutant le paramètre *source*. Les opérations évoquées ici sont les "vraies" opérations de l'entité mais aussi celles de consultation/modification des caractéristiques (*getX*, *setX*, *addX*, ...). Ces dernières opérations sont généralement la cause d'une projection (des méta-modèles) vers un support d'implémentation. Elles n'apparaissent pas dans une vue conceptuelle. Toutefois, dans un but pédagogique, nous les indiquons explicitement sur la figure 49 pour la classe *Task_Process_25* qui constitue le traitement d'adaptation *Task(Process)* "à un moment donné".

4.2 Le service

Nous avons vu les classes générées qui constituent la surcouche d'adaptation et les classes de CAST (*Adapter*, *Filter* et *ProcessingObject*) qui définissent les caractéristiques communes de ces classes générées. La coordination des adaptateurs, des filtres et des objets traitements est gérée par un service (d'adaptation). Il gère un ensemble d'adaptateurs, un ensemble de droits d'accès, et les traitements d'adaptation qu'il permet de changer. Chaque ensemble est géré par une table.

4.2.1 Les tables

Les figures 50, 51 et 52 montrent le fonctionnement des tables d'adaptateurs, des droits d'accès et des projections/traitements d'adaptation.

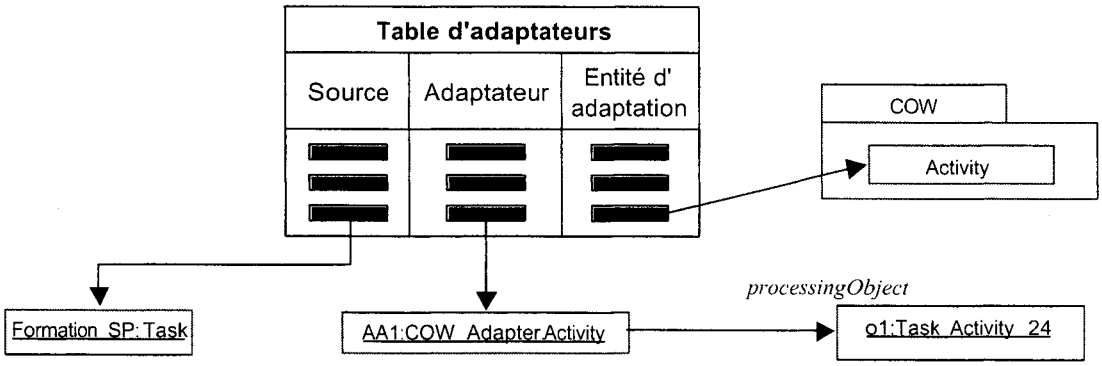


figure 50. Une table d'adaptateurs

Une table d'adaptateurs permet de connaître l'adaptateur existant pour une source donnée pour un type d'adaptation particulier. Par exemple une *tâche* peut avoir trois adaptateurs différents dans le système COW : elle peut être vue comme un *Process*, une *Activity* ou un *WorkItem*. C'est la fonction *adapt* qui consulte cette table. Si une source n'a pas encore été adaptée, la table crée un adaptateur selon les projections contenues dans la table de projection.

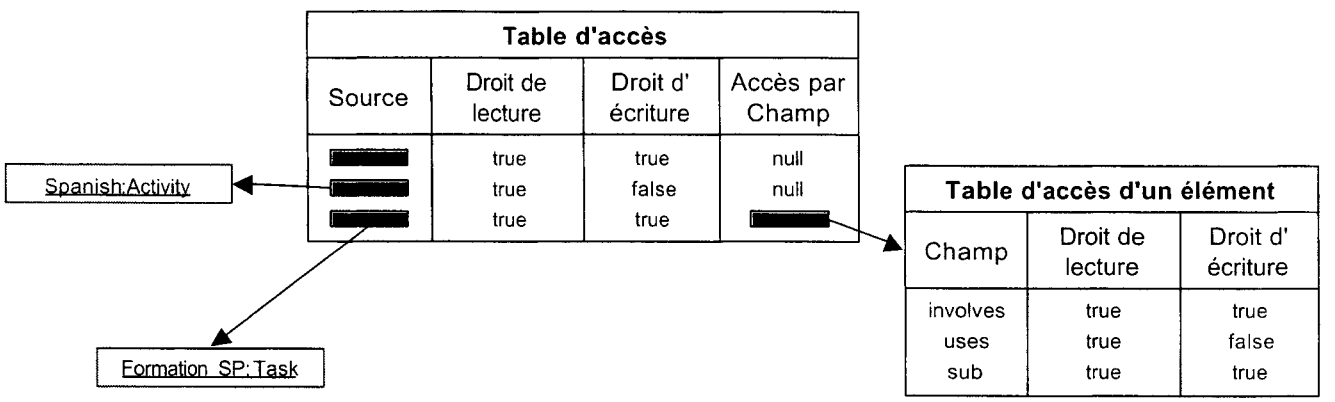


figure 51. Une table de droits d'accès

La table des droits d'accès est utilisée par les filtres. Elle leur indique si un objet est accessible et si oui, quelles sont les caractéristiques accessibles.

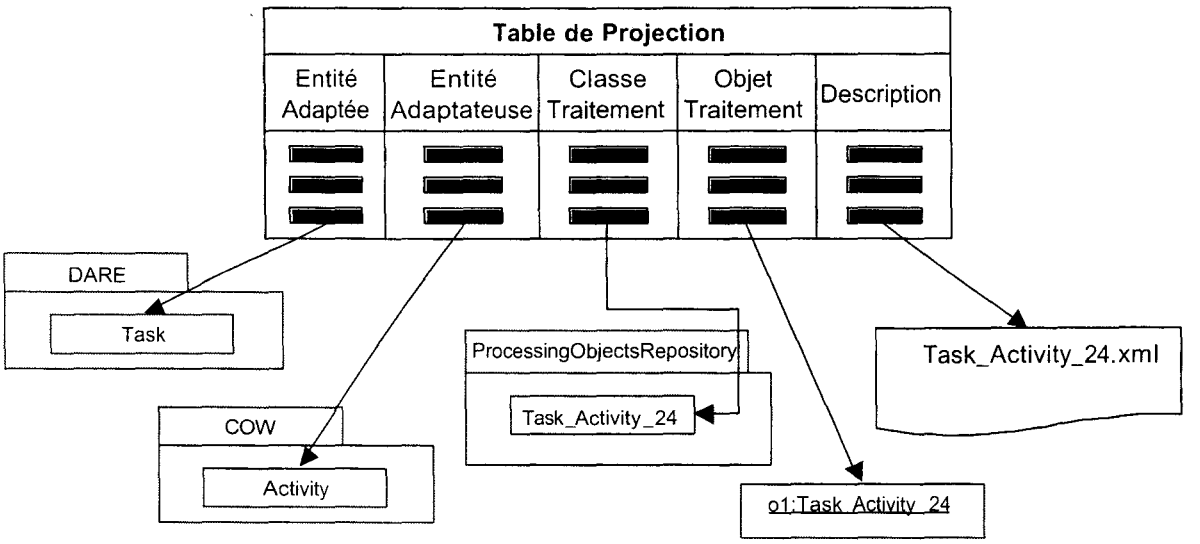


figure 52. Une table de projection

La table de projection est utilisée par l'opération *adapt* lorsque celle-ci doit créer (ou retrouver) un adaptateur pour un objet et un type de projection donné. La table lui indique quel objet traitement utiliser. Elle dispose de fonctions de modification des traitements associés, mais aussi des liens conceptuels. Ainsi il est possible de rajouter ou de retirer un type d'adaptation pour un concept. Dans ce cas, les adaptateurs (concernés) déjà existant demeurent. Il n'est par contre plus possible de créer de nouveaux adaptateurs de ce type. Nous avons fait le choix de garder les adaptateurs déjà existants pour ne pas à se poser le problème des répercussions d'une suppression.

4.2.2 Le service d'adaptation

Le service d'adaptation, qui coordonne l'ensemble de l'adaptation, est donc la composition de ces trois tables. La modélisation précise du service complet resitue chacun des composants d'un service d'adaptation (cf. figure 53).

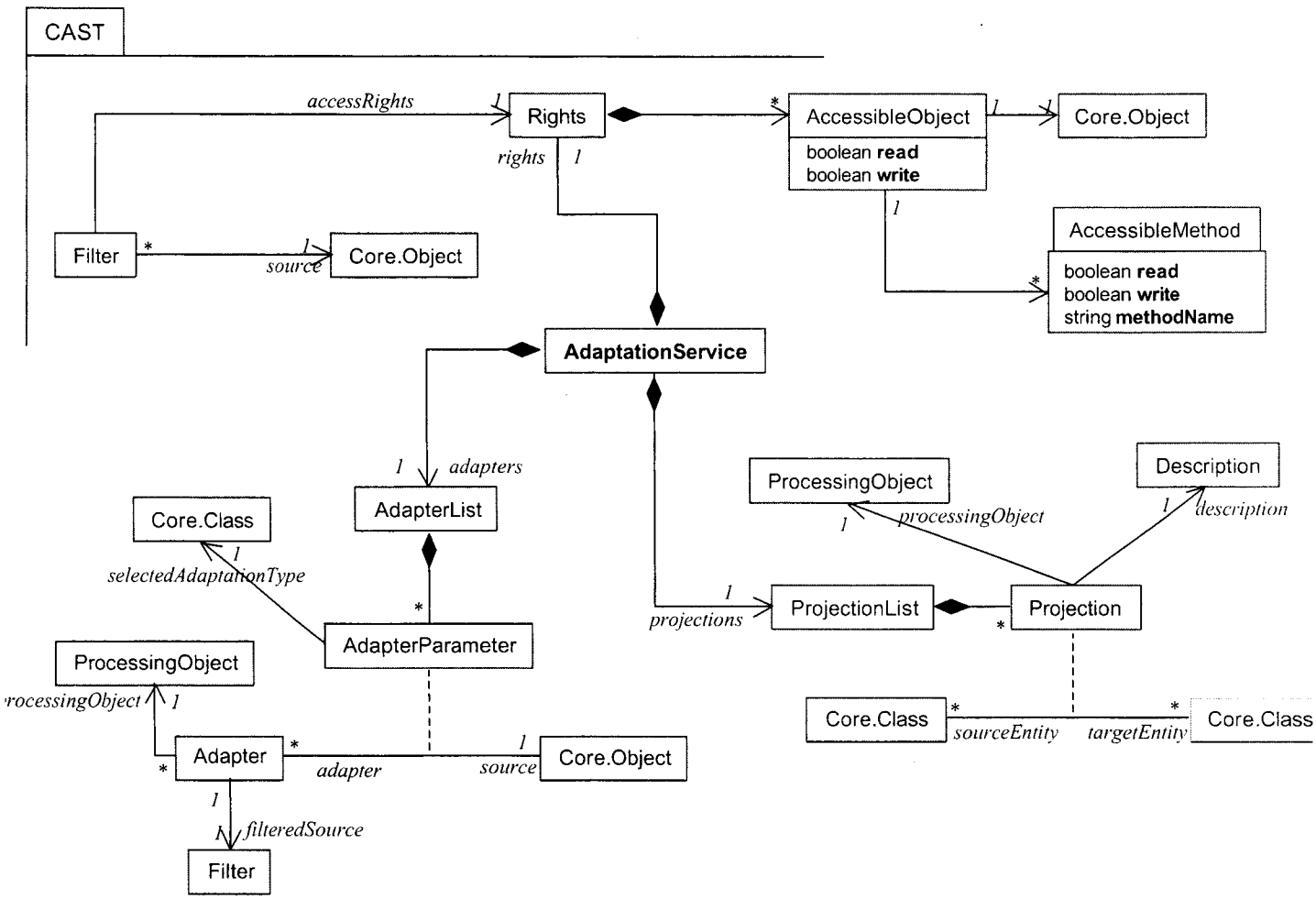


figure 53. Le service d'adaptation

Les classes *AdapterList*, *Rights* et *ProjectionList* correspondent aux tables précédentes.

4.3 L'ensemble

La figure 54 récapitule le schéma de génération. Les liens d'adaptation génèrent un paquetage d'adaptateurs et un paquetage de filtres pour chaque méta-modèle. Ils utilisent le paquetage CAST qui constitue le noyau du service d'adaptation. Ce paquetage est commun à tous les AS (et ne fait donc pas partie des éléments générés lors de la génération d'un AS).

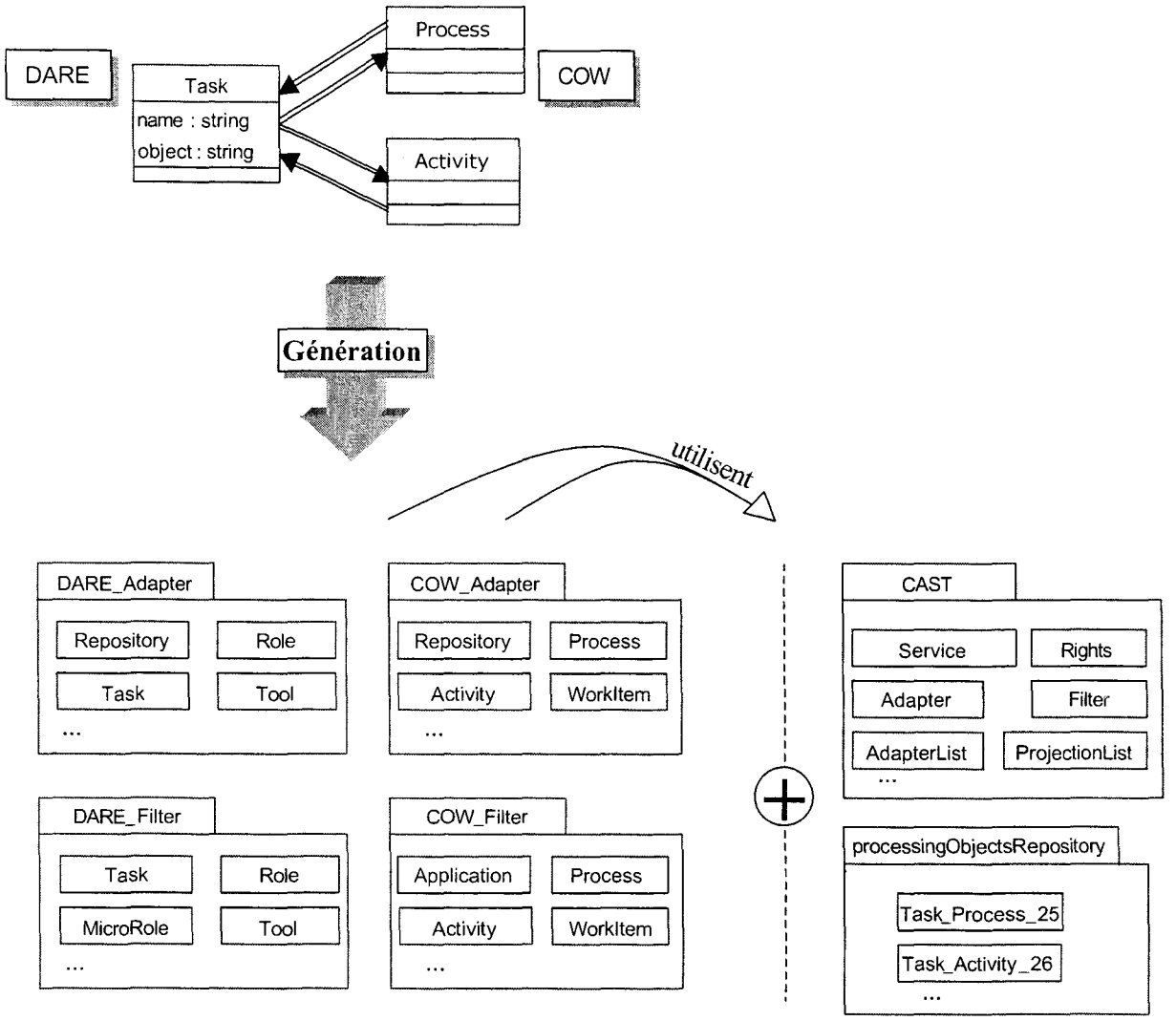


figure 54. Paquetage de base et paquetages générés

4.4 Séquences

Pour compléter notre modèle conceptuel, nous terminons avec des diagrammes de séquence. Ceci apporte une précision sur le fonctionnement du service d'adaptation. Deux exemples particuliers nous semblent suffisants pour comprendre le fonctionnement global du service d'adaptation : l'accès au référentiel et l'accès à une caractéristique d'un élément.

4.4.1 Accès au référentiel

Nous prenons comme exemple une requête de HLC pour connaître les tâches importées (issues de LECS). Ceci se traduit par l'invocation de la méthode *allInstancesOf_Task* du référentiel-adaptateur DARE(COW). Ce dernier demande au service, auquel il est rattaché, toutes les tâches adaptables (*getAllAdaptersOf*). Pour cela, le service demande à la table de projection quelles sont les entités dans COW qui se projettent en *tâche*. Le service demande ensuite à la table d'accès d'avoir les *process*, les *activity* et les *workItem* accessibles. Ensuite il les adapte. L'adaptation est effectuée par la table des adaptateurs qui crée un adaptateur pour chaque source n'ayant pas déjà été adaptée.

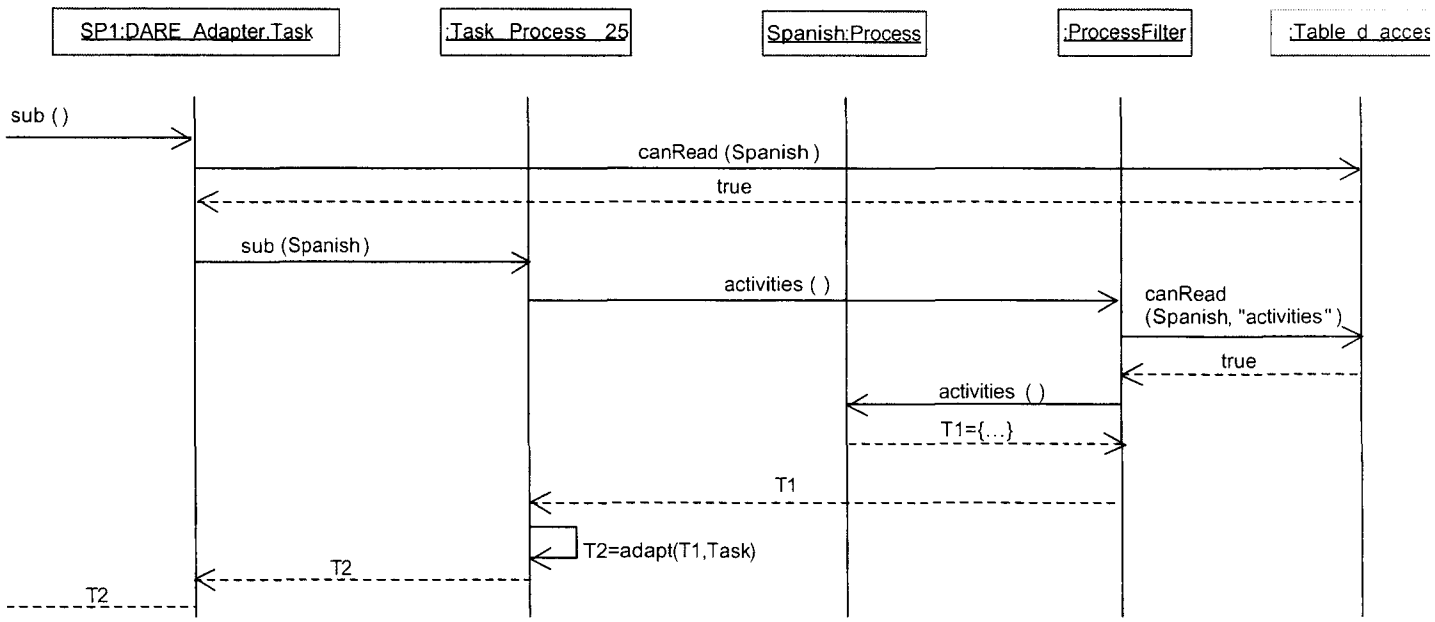


figure 55. Demande des tâches importées

4.4.2 Accès à une caractéristique

Sur une tâche *Task[Spanish]* (*SP1:DARE_Adapter.Task*) qui a été reçue par l'invocation précédente sur le référentiel, sont réclamées les sous tâches (*get_subTasks*). L'adaptateur demande dans un premier temps, si *Spanish* est accessible en lecture. Comme il l'est, l'adaptateur demande à l'objet traitement qui lui est associé (*:Task_ Process _25*) la même "question" en lui passant la source "filtrée". Celui-ci va traduire cette requête en demande des activités sur la source filtrée. L'opération `activities()` de la source filtrée vérifie qu'*activities* est accessible en lecture. Comme c'est le cas, elle renvoie les activités. L'objet traitement adapte ces activités en tâches car c'est le type de retour de la méthode.

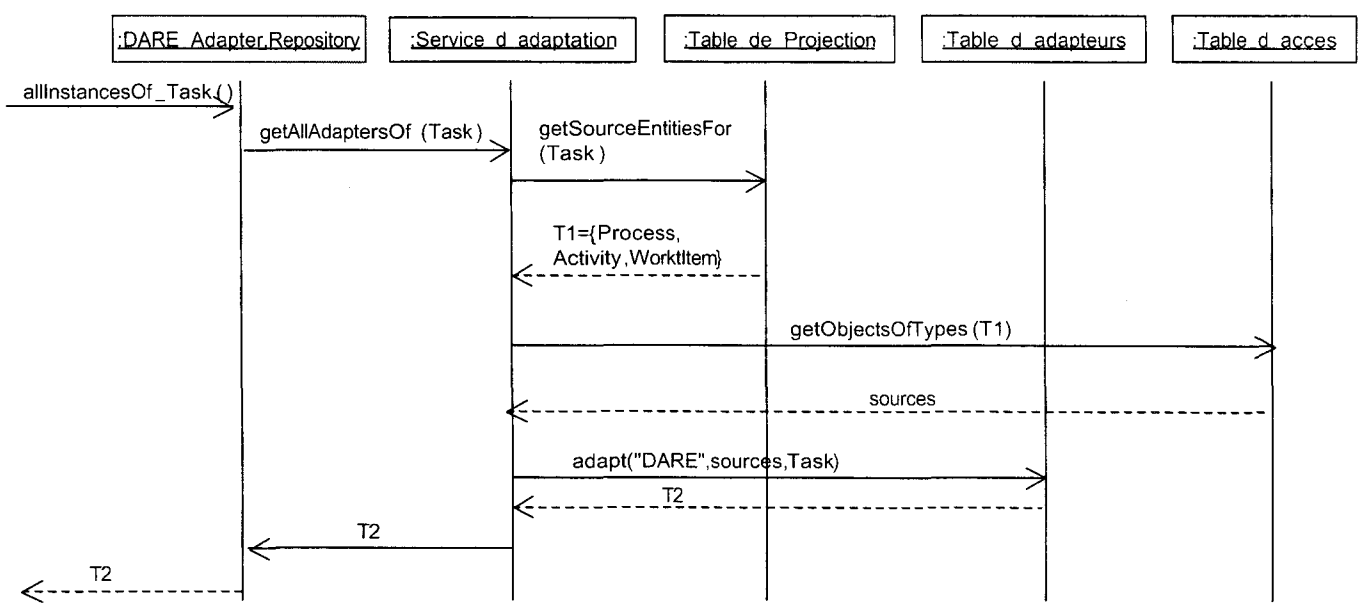


figure 56. Accès à une caractéristique

5 Conclusion

Devant l'absence actuelle de solution technique pour faire coopérer des méta-groupware, nous proposer une solution : CAST. Elle est basée sur la notion d'adaptateur. Notre objectif est de fournir un outil pour construire rapidement des adaptateurs pour ce type d'architecture. L'utilisation de cet outil commence par la définition, effectuée par des informaticiens, de liens d'adaptation entre les deux méta-modèles. Ces liens génèrent ensuite un service d'adaptation pour les deux systèmes. Une fois connecté aux deux systèmes, le service permet aux utilisateurs d'un système de voir les modèles de l'autre système et d'en référencer des éléments. L'adaptation se fait aussi pour les instances de modèles.

La conception de cet outil a demandé plusieurs travaux. D'abord la définition d'un formalisme graphique pour exprimer des liaisons d'adaptation. Nous avons ensuite conçu un pattern de service d'adaptation (pour ne pas être attaché à un support particulier). Avec l'éclaircissement sur les notions de concept de type et concept d'instance (cf. chapitre 3) le formalisme et le pattern utilise la particularité de la structure à trois niveaux pour accélérer la conception d'un AS.

La validation de notre approche passe nécessairement par des tests d'une application de CAST sur un support d'implémentation. Il nous faut donc implémenter le pattern de service d'adaptation et par conséquent choisir un support pour l'accès informatique des méta-modèles et modèles.

Chapitre 5

Support de représentation des modèles et de réalisation de CAST : Le Meta-Object Facility

La mise en place des services d'adaptation de modèles et de leurs instances est une solution adaptée à la collaboration de systèmes flexibles. La construction d'un environnement de conception rapide et de prototypage pour ces services permet de réduire le problème du nombre élevé de services d'adaptation, et rend plus efficace notre proposition. Nous avons insisté dans le chapitre 2 sur l'importance de la standardisation : un des problèmes principaux à l'origine de notre proposition est la difficulté d'avoir un méta-modèle standard pour un domaine d'information particulier. Notre proposition nécessite toutefois de se reposer sur des standards de plus bas niveaux. L'Object Managment Group (OMG) a défini en 1995 le Meta-Object Facility (MOF) [OMG 00][LEP 00]. Il a pour but de fournir un standard de représentation de tous types de modèles. C'est une des seules initiatives de ce type qui a des chances de réussir. C'est sur le MOF que nous allons réaliser CAST. Le principal inconvénient du MOF est de ne pas avoir la possibilité d'instancier ces modèles et donc de lier ces derniers avec leurs instances. Nous évoquons dans ce chapitre nos travaux sur une formalisation de ces liens.

L'objectif de ce chapitre est triple :

- Présenter le support avec lequel nous réalisons CAST et expliquer ce choix (section 1.4). L'application de CAST sur le MOF clotûre ce chapitre (partie 5). La présentation précise du MOF est l'objet de la partie 3.
- Tenter de clarifier l'utilité du MOF "dans l'absolu" (partie 2).
- Illustrer les liens modèles et leurs instances de manière concrète (i.e. sur le support MOF). Etant spécifique à l'approche par méta-modèle, cet aspect de notre solution est primordial (partie 4).

1 Objectifs du MOF

Le MOF a été créé par le consortium OMG. Dans cette partie, nous verrons le contexte des origines du MOF et l'orientation prévue pour celui-ci. Nous donnons aussi un premier aperçu des spécifications avant de voir des exemples précis d'utilisation.

1.1 L'OMG et l'architecture OMA

Le consortium OMG a été créé, au début des années 90, sous l'impulsion d'acteurs importants du monde informatique (3Com, Sun microsystems, Hewlett-Packard, Unisys Corporation, ...). Ce consortium s'est fixé comme objectif de supprimer le problème d'interopérabilité entre applications industrielles [OMG 02]. Les premiers travaux entrepris sont la définition d'une architecture, nommée OMA (Object Managment Architecture). Au cœur de celle-ci se trouve le bus CORBA. Il s'agit d'un bus à objets répartis qui permet à des applications, issues de langages et de systèmes d'exploitation différents, de communiquer entre elles. L'aspect communication/technique du problème de l'interopérabilité est donc réglé. L'intégration de système légataire, la ré-utilisation de composants d'application, l'évolution du code sont, de plus, améliorées grâce à l'utilisation du paradigme objet dans les spécifications du bus CORBA.

Plus les différences entre applications s'amenuisent (i.e. plus l'environnement dans lequel se trouvent les applications est homogène) plus il est facile de les faire interopérer. Le bus CORBA se voit ainsi associer différents éléments fournissant les briques de bases à la conception d'applications distribuées : les services communs, les utilitaires communs et les interfaces de domaines [GEI 97b] communs (CORBA Common Object Services) offrent les fonctions systèmes de base telle que la persistance, la sécurité ... Les utilitaires communs (CORBA Facilities) offrent, quant à eux, des fonctionnalités plus abstraites comme la gestion des interfaces utilisateurs. Enfin les interfaces de domaine (Domain interfaces) proposent des "objets CORBA réutilisables" qui décrivent des informations de domaines bien spécifiques tels que le monde médical, le marché des télécommunications...

Le travail de mise en place de liens entre deux applications commence à un niveau de modélisation, où se trouve la description de l'architecture des applications, la façon dont elles sont déployées ... En 1995, un effort est fourni par l'OMG pour définir un formalisme graphique standard de modélisation. Le résultat sera la définition des spécifications d'UML en 1997[47]. Outre les échanges entre équipes de développement, UML sert aussi à la génération de code [UML 02][48].

1.2 Le MOF et ses applications

Si les problèmes liés à l'hétérogénéité ont été résolus à un niveau technique grâce au bus CORBA, ils persistent en terme d'interopérabilité au niveau sémantique. Dans le chapitre 2, nous avons vu que cette problématique n'a jamais trouvé de réponses réelles. Plutôt que proposer une "impossible" solution parfaite, l'OMG décide de fournir les éléments qui interviennent souvent dans les solutions d'interopérabilité sémantique : les méta-données.

Le Meta-Object Facility est défini en 1997. Il est donc créé en même temps qu'UML, dont il est un "dérivé", mais passe inaperçu de la communauté des développeurs d'applications. Le MOF est destiné, dans un premier temps, à définir la structure (i.e. l'interface IDL) des objets CORBA représentant des méta-données, c'est-à-dire des méta-modèles et des modèles. Le noyau du MOF est ensuite, en 1999, utilisé pour définir la structure de documents XML représentant des méta-données. Ceci donne les spécifications : XML-based Meta-data Interchange (XMI [XMI 02]). Ce support rencontrera bien plus de succès. Il sera notamment utilisé comme format de persistance standard pour les modèles UML.

Définir un standard de "représentation mémoire" et de persistance de méta-données est assez innovant mais aussi incompris. Nous décrivons pour cela, dans la partie suivante (cf. 0), des situations qui illustrent l'intérêt d'une telle initiative. Un rapide aperçu des spécifications MOF est utile pour étudier ces situations.

---

[47] Booch, Jacobson et Rumbaugh, auteurs respectifs des trois méthodes orientés objet célèbres, sont des acteurs importants dans la définition d'UML.

[48] Depuis la version 1.1, les spécifications d'UML intègre d'ailleurs des règles de génération d'interface IDL

1.3 Les spécifications du MOF : un premier aperçu

Le principe des spécifications du MOF (à partir de la version 1.3) est de définir un formalisme de description de méta-modèles (i.e. un méta-méta-modèle) et un support d'instanciation abstrait. Le formalisme choisi est un sous-ensemble d'UML. Deux supports d'instanciation "concrets" sont associés à ces spécifications : Corba et XML. Les modèles (instances de méta-modèle) sont manipulés soit à travers des objets Corba, soit à travers des documents XML.

Les supports Corba et XML suivent les règles définies par le support abstrait[49]. Ils forment un ensemble de règles de génération d'interfaces IDL pour le premier et de DTD pour le second. Les interfaces IDL sont les types des objets CORBA représentant des modèles, et une DTD est la structure des documents XML représentant des modèles.

Quand un concepteur est amené à créer un nouveau méta-modèle, l'OMG lui propose de définir celui-ci à travers le formalisme MOF (UML allégé). L'aspect standard de ce dernier, permet au concepteur d'échanger plus facilement son méta-modèle avec d'autres concepteurs, et d'utiliser un plus grand nombre d'outil de méta-modélisation. Lorsque le concepteur est conduit à gérer des instances du méta-modèle précédent dans un système (comme des modèles de processus), l'OMG lui offre un support distribué (Corba) et un support persistant (XML). Nous verrons dans la partie 2, qu'un outil MOF génère les "éléments" nécessaires pour la gestion de modèles dans ces deux supports. Si ceux-ci ne conviennent pas au concepteur, il peut créer son propre support (EJB, Java/RMI, .Net, ...) à partir du support abstrait. Il perd néanmoins la compatibilité avec d'autres outils MOF (par exemple, un lecteur générique de méta-données MOF/Corba[50]).

1.4 Pourquoi choisir le MOF pour la réalisation pratique de CAST ?

Un système doit nécessairement avoir une base commune avec un autre système pour dialoguer avec lui. Sans rentrer dans une réflexion sur le niveau d'abstraction de cette base, on peut dire qu'en général, l'utilisation d'un middleware (ex : Corba, WebServices, .NET, RMI, ...) permet pour beaucoup de systèmes de respecter cette contrainte. Pour les méta-groupware, et plus généralement pour les méta-systèmes, le dialogue va contenir des informations représentant des modèles. Si, nous l'avons vu, il est rarement possible d'avoir un formalisme de modélisation standard (même pour de petits domaines d'informations), il est par contre possible d'avoir un format standard de représentation des modèles. Le pouvoir d'expression des formalismes de modélisation n'en est pas limité pour cela.

Dans notre contexte d'adaptation de modèles, il est de ce fait évident qu'il nous faut un tel format standard (la partie 2 illustre cette affirmation). Après la tentative avortée de l'EIA (i.e. CDIF), le MOF est aujourd'hui le seul format standard de représentation de modèles. Pour cette raison et parce qu'il nous faut utiliser un tel format, le MOF constitue notre élément de base à la réalisation pratique de CAST.

2 Utilité du MOF

L'intérêt de la standardisation des méta-données n'est pas évident a priori. Si, dans ses débuts, le MOF est passé pour inutile, c'est qu'aucune argumentation solide n'a été présentée. Au cours de nos travaux, nous avons pu déterminer quels étaient les réels bénéfices du MOF. Les trois sections suivantes présentent chacune une situation où apparaissent ces bénéfices. La quatrième et dernière section n'est pas un quatrième exemple mais plutôt une description de certains mécanismes du MOF dont l'utilité apparaît dans des cas très spécifiques.

---

[49] De la même manière qu'en langage orienté objet, une classe reprend la structure d'une classe abstraite si elle en hérite [EDOC 2002].

[50] Inexistant à l'heure actuelle mais c'est un projet du DSTC [DST 02]

2.1 Transformation de modèles

2.1.1 Contexte

Nous sommes dans le cas où il y a une volonté de transformer des modèles Merise en modèles UML. Un certain nombre d'outils Merise existent (ex : WinDev, PowerAMC, Win'Design). En plus de structurer la démarche du développeur, certains outils permettent, à partir de la définition d'un modèle, de générer les scripts de création de tables de données (et parfois le code d'implémentation – dans un langage particulier – concernant la manipulation de ces données). D'un autre côté, le formalisme de modélisation UML prend une place de plus en plus importante. Sa présence dans les ateliers de conception de logiciels est croissante. L'outillage qui lui est associé commence à être conséquent : Rational Rose propose, par exemple, des générations de code en Java, VB, Corba... UML est souvent associé au travers de ces outils, à des technologies récentes "d'implémentation" comme les EJB. Pour pouvoir échanger plus facilement leurs modèles avec d'autres équipes de développement et profiter d'un outillage plus moderne, certains utilisateurs de Merise peuvent élire UML comme nouvel outil de modélisation. Dans ce cas, ils désirent sûrement avoir une version UML des modèles Merise qu'ils ont conçus auparavant, de manière à les ré-utiliser ultérieurement. La perspective de disposer d'un outil de transformation de modèles Merise en modèles UML devient séduisante.

2.1.2 Conception d'un transformateur

En tant que développeur aguerri, nous décidons d'implémenter le transformateur décrit précédemment. Le scénario de transformation que nous élaborons est décrit sur la figure 57.

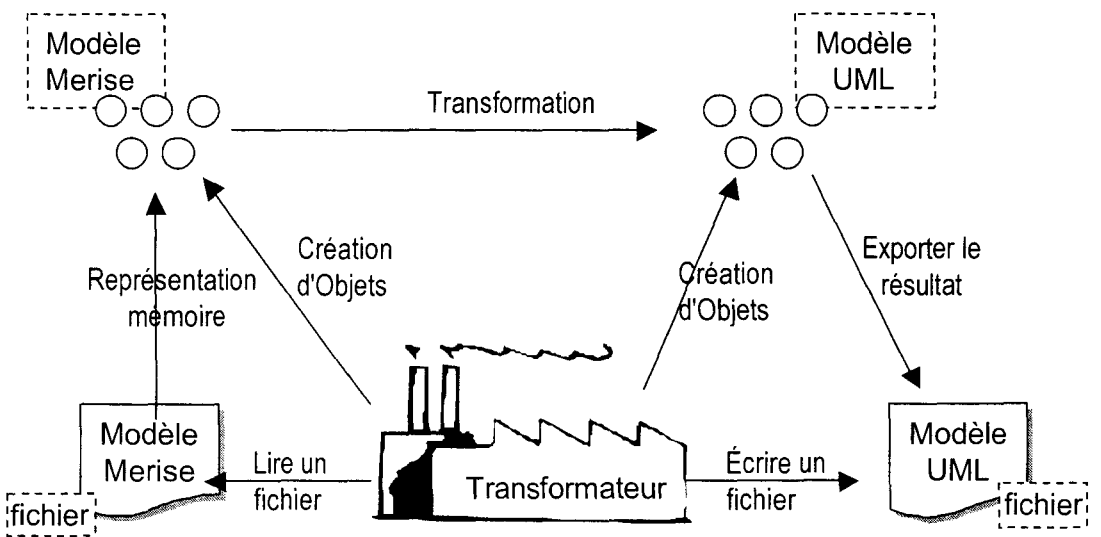


figure 57. Scénario de transformation

Dans un premier temps, le transformateur lit un fichier contenant un modèle Merise. Il crée une représentation mémoire de ce modèle à travers des objets (Java, C++ ou autres). La transformation, c'est-à-dire l'opération principale, s'effectue ensuite : à partir des précédents objets, création de nouveaux objets qui représentent cette fois le modèle UML correspondant. La lecture des valeurs de ces derniers objets permet l'écriture du modèle UML dans un fichier.

Au travers de ce scénario, on peut différencier quatre nécessités :

1. Connaître les concepts de modélisation de Merise et d'UML.
2. Connaître les formats de sauvegarde des outils Merise et UML utilisés.
3. Trouver une structure pour les deux types de représentation mémoire.

4. Établir les "règles" de transformation.

2.1.3 Standardiser l'accès aux modèles

D'un point de vue pratique, les deux étapes les plus fastidieuses sont les étapes 2 et 4.

L'implémentation du moteur de transformation est le cœur du programme. Si les règles de transformation peuvent sembler simples aux premiers abords (ex : une entité Merise donne une classe UML), elles se complexifient rapidement lorsque chaque caractéristique est pris en compte (ex : une relation Merise peut engendrer une classe d'association ou non, détecter les relations d'agrégation, ...).

La deuxième étape est l'étude des formats de sauvegarde UML et Merise. En supposant que tous les outils UML ont le même format de sauvegarde (même hypothèse pour les outils Merise), celui-ci ne se déduit malheureusement pas directement des concepts UML. Il faut donc étudier la "grammaire" adoptée : soit elle est documentée, soit il faut la déduire de fichiers existants. C'est sur ce point qu'intervient le Meta-Object Facility : il standardise le format de sauvegarde des méta-données. Parce qu'il est impossible d'avoir un format universel pour tous les types de méta-données existantes, le MOF standardise la structure des grammaires : il définit des règles de génération de DTD. Ainsi, avec la description des concepts d'UML en MOF et grâce aux règles précédentes, la DTD (i.e. la grammaire) qui lui est associée est facilement déductible. L'étape 2 est ainsi grandement accélérée.

Le MOF peut aussi simplifier l'étape 3. En effet, celui-ci définit des règles de projection vers le langage IDL. Pour le méta-modèle Merise, par exemple, les règles définissent les interfaces IDL correspondantes à chaque concept. Grâce à un pré-compilateur IDL, peuvent être générés les classes Java ou C++ (ou d'autres encore) correspondantes. Les prochains outils MOF devront être capables de lire un document XML représentant un modèle Merise et créer les objets Corba correspondants. Une telle caractéristique permettrait au développeur du transformateur de se focaliser essentiellement sur les règles de transformation.

Dans le développement d'un traducteur de modèles, la standardisation du format des modèles allège considérablement le travail du développeur.

2.2 Groupement de systèmes et médiateurs

2.2.1 Rappel

Dans l'exemple de l'entrepôt de données présenté dans le chapitre 2, l'environnement intègre différentes sources de données. Les demandes/requêtes des utilisateurs sont envoyées à un médiateur. Ce composant logiciel adresse aux sources concernées des (sous-)requêtes dont les réponses lui permettront de construire le résultat final. Cette redistribution du traitement est guidée par une description des liaisons entre les structures de données existantes dans les différentes sources. La description, saisie par l'administrateur de l'environnement, est gérée par un composant connexe au médiateur.

2.2.2 Méta-données

Les liaisons entre les modèles de données se font dans un langage "unique". Une traduction des modèles existants, exprimés dans leur formalisme originel, vers ce langage est donc nécessaire. Si cette opération pouvait parfois se faire manuellement auparavant, aujourd'hui elle est généralement automatisée dans les solutions plus récentes. Deux méthodes existent :

- Soit la traduction se fait par un module séparé, et le résultat est intégré dans le référentiel de modèles par l'administrateur [SHE 90].
- Soit le composant gérant les liaisons s'adresse directement aux différentes sources de données pour récupérer leurs modèles et notifie l'administrateur après les avoir traduites et intégrés [BON 98]. On sous-entend ici que tout changement de modèles sera automatiquement répercuté dans le référentiel de modèle *unique*.

Dans ces deux méthodes, le besoin de standardiser l'accès aux méta-données se fait sentir. La première, comme dans la section 2.1, peut se satisfaire d'un support statique comme XML. La deuxième méthode, par contre, tirera plus d'avantages à utiliser un support dynamique comme Corba : des objets Corba représentent les modèles de données de chaque source, et le service (Corba) de notification peut être utilisé conjointement pour prévenir de tout changement, en indiquant précisément à chaque fois quels méta-objets ont été modifiés. Nous retrouvons d'ailleurs ce genre de méthode dans les récentes spécifications du CWM précédemment évoqué (cf. chapitre 3). Dans la conception de tels environnements, pour chaque intégration d'un nouveau type de sources de données, il faut ajouter au composant gérant les liaisons la capacité d'accéder aux nouveaux types de méta-données. Si cette source est compatible MOF, la description des concepts de modélisation, qu'elle utilise, permet aux concepteurs de déduire facilement les interfaces IDL des objets CORBA représentant ses modèles de données. Ce gain de temps notable dans la conception d'environnement de groupement de systèmes est un autre bénéfice de la standardisation des méta-données.

2.3 Conception d'un nouveau système d'information

Le MOF n'a pas comme seul avantage de faciliter l'accès aux méta-données. Toutes les spécifications définies par l'OMG sont généralement suivies par des produits les implémentant. Le MOF ne déroge pas à cette règle : les produits MOF sont des applications qui permettent, pour un méta-modèle donné, de générer la DTD ou les interfaces IDL correspondantes. Ces dernières sont souvent accompagnées d'une implémentation (généralement en Java) qui constitue un serveur de méta-données. Cette caractéristique est bien sûr intéressante pour la conception des systèmes flexibles que nous étudions depuis le chapitre 3 où, pour rappel, une structuration en trois niveaux est une implémentation efficace (M2, M1, M0).

Lors de la construction d'un tel système, chaque niveau est à implémenter. Après avoir décrit les concepts de modélisation en MOF, un outil MOF va générer les deux couches les plus abstraites (M2 et M1). Le programmeur n'a donc plus qu'à développer la couche de contrôle d'exécution (M0). Pour les systèmes récemment créés dans notre laboratoire (DARE et COW), cette couche s'est révélée la plus fastidieuse à implémenter : elle doit respecter les modèles définis (implémenter des mécanismes de look-up, d'invocation,... spécifique à une double instanciation) et prendre en compte l'environnement dans lequel le système va s'exécuter (i.e. ressources disponibles, politique d'accès à ces ressources, ...). Néanmoins les couches M2 et M1 représentent aussi une charge importante de travail : contraintes à respecter (que l'on peut exprimer à l'aide contraintes OCL dans le cas du MOF), service de persistance, interface de manipulation, ... La génération des couches M2 et M1 par un outil MOF[51] est donc un gain de temps dans le développement de système.

Il est important de noter que la génération est une caractéristique qui n'est qu'en partie due au MOF : l'OMG n'a défini que des règles de génération. Ce sont les outils qui génèrent les interfaces IDL et l'implémentation de ces dernières. De plus, d'autres outils de méta-modélisation offrait déjà ce type de génération (une partie des outils méta-CASE comme METAGEN [REV 97]). L'apport du MOF est, comme on l'a vu, de fournir un accès standard, mais aussi de mettre en avant la méta-modélisation comme structure à un système et de favoriser les outils de méta-modélisation (ce qui offre ainsi un choix plus varié pour les développeurs).

2.4 Un support puissant de méta-modélisation

Un des derniers bénéfices du MOF apparaît dans les caractéristiques structurelles de celui-ci. Ce support est en effet très puissant : héritage multiple, mécanismes de réflexivités, fonction de copie, règles de génération pré-établies... La gestion de types ou d'informations constitue deux exemples qui mettent en avant les différentes fonctionnalités du MOF.

[51] Tout n'est pas généré. Par exemple aucun outil ne génère d'interface de manipulation. Néanmoins les lacunes devraient être vite comblées.

2.4.1 Gestion de types

Les services communs de l'architecture OMA contiennent généralement des référentiels d'objets CORBA représentant des types d'informations : l'Interface Repository (IR) permet de connaître l'interface IDL d'un objet CORBA, le service de courtage (Trader) contient la description de services proposés par des objets CORBA, ... Un des scénarios d'usage provenant des spécifications du MOF est le remplacement de l'IR actuel par une version MOF. L'utilisation du MOF comme support au référentiel d'interfaces IDL permettrait :

- de lui ajouter la capacité de mise à jour (qu'il n'a pas),
- en cas de modification du méta-modèle IDL, il faut modifier l'ensemble des interfaces IDL de l'IR de manière à en répercuter les modifications. En utilisant le MOF, ceci se ferait automatiquement en régénérant les interfaces IDL à partir de la nouvelle version du méta-modèle [52],
- avoir un lien sur des objets CORBA représentant les concepts du méta-modèle IDL.

Dans ce scénario, est envisagée, dans un futur proche, la présence d'un service de passerelles d'interfaces. Cette caractéristique n'est pas liée à la puissance du support MOF, mais il est intéressant d'en exposer les traits principals. Dans un système à grande échelle, certains services au fil du temps disparaissent. Il faut proposer une alternative aux parties du système, clientes de ces services. L'idée est de procurer un service de passerelle, qui pour un type de service donnée "ancien" fournit le type de service "nouveau" correspondant (un mécanisme d'instanciation sera bien sûr présent[53]). Ce service de passerelle sera peut-être un des éléments générés par les futurs outils MOF.

Enfin Le MOF peut constituer un support intéressant à des standards comme PICS ou RDF pour la description du contenu de documents (*content-based* [PIC 97][RDF 99]). Son utilisation permettrait d'avoir des descriptions de documents issus d'un méta-modèle spécialisant le méta-modèle "par défaut" (standard, comme RDF). On pourrait avoir ainsi des descriptions plus adaptées à certains types de documents. Les éléments supplémentaires de description pourraient être découverts et intégrés par un outil, prévu pour le méta-modèle "par défaut", si celui-ci utilise les mécanismes d'héritage et de réflexivité du MOF.

2.4.2 Gestion d'information

Dans son support dynamique, le MOF est destiné à la conception de référentiels de modèles (appelés aussi méta-données). Parce qu'une méta-donnée est aussi une donnée, c'est de manière logique que les outils MOF peuvent servir dans la conception de référentiels de données. Par exemple, un modèle UML/MOF dont les classes sont Livre, Client, Achat, Commande, ... "envoyé" à un outil MOF, permet de générer le référentiel d'informations du logiciel de gestion d'une librairie. De base, un outil MOF offre pour ce genre de construction, une implémentation de type distribué, un format de persistance, une compatibilité avec des outils de lecture de référentiels MOF (utile pour du monitoring) et, à l'exécution, des liens sur les objets représentant le modèle de ces informations.

3 Le MOF

Dans cette partie, nous présentons la partie centrale du MOF, c'est-à-dire le méta-méta-modèle. Les deux sections qui suivent décrivent respectivement les supports XML et CORBA du MOF. Les outils MOF existants constituent l'objet de la dernière section.

---

[52] Il faudrait au préalable modifier les règles de génération pour prendre en compte les modifications du langage IDL.

[53] Les liens entre anciens et nouveaux services sont à saisir par les développeurs du système. Ils indiquent aussi l'emplacement des mécanismes d'instanciation.

3.1 Le méta-méta-modèle MOF

3.1.1 Principe

Le méta-méta-modèle MOF est un sous-ensemble du méta-modèle UML. Nous reprenons ici les principaux concepts de ce formalisme mais d'un point de vue *méta-modélisation*. Dans ce but, les concepts décrits dans un méta-modèle seront désignés par le terme *entité*[54], et les éléments du méta-méta-modèle MOF par le terme *méta-entité*.

La figure 58 présente les principales méta-entités du MOF ainsi que leurs principales relations. Un méta-modèle est défini à l'aide de la méta-entité *Package*. Par exemple, le paquetage Merise contient les entités *Entité*, *Relation*, ... Ces entités sont décrites grâce à la méta-entité *Class*. *Association* permet de définir les relations existantes entre les entités. Les associations ne peuvent être que binaire. Il ne peut y avoir non plus de classe d'association. Une entité est définie par un ensemble d'attributs (méta-entité *Attribute*), d'opérations (méta-entité *Operation*) et de références (méta-entité *Reference*). Une référence est la connaissance pour une entité de sa participation dans une association. Par exemple, si l'entité A est associée avec l'entité B et que A dispose d'une référence R pour cette association, les valeurs de R pour *a*, une instance de A, seront toutes les instances de B associées à *a*. La méta-entité *DataType* permet de définir des types de valeur non-objet (il peut s'agir de types prédéfinis du MOF - string, short, ... - ou des types propres au méta-modèle). La méta-entité *Exception* permet de décrire les exceptions liées aux opérations.

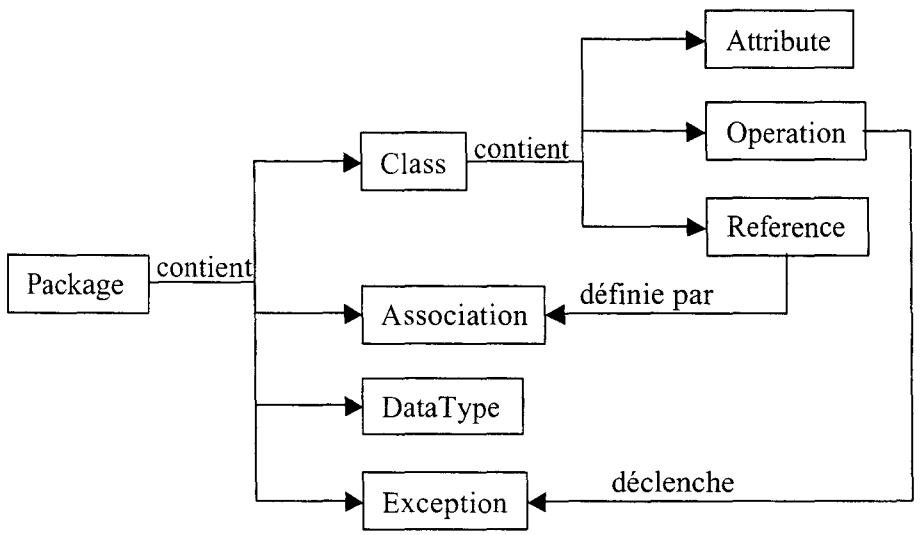


figure 58. Principales méta-entités du MOF

3.1.2 Exemple : le méta-modèle de DARE

La figure 59 est la description MOF du méta-modèle de DARE. Celui-ci nous servira d'exemple pour les support XML et Corba. Il est identique à celui-ci présenté dans le chapitre 3. Les éléments d'UML supprimés pour le MOF, comme les classes d'associations, n'étant pas utilisés dans ce "modèle", la transition UML→MOF ne demande aucune transformation.

---

[54] Les concepts d'instance ne sont pas représentés comme tels dans un méta-modèle MOF. Si le concepteur fait apparaître le CI *Activity* de DARE dans un méta-modèle MOF, il a le même statut (d'entité) que *Task*. Aucune caractéristique du MOF n'est présente pour indiquer directement qu'il s'agit d'un CI. C'est la compréhension du méta-modèle (avec, par exemple, un lien isARealisationOf entre *Task* et *Activity*) qui permet à un concepteur de le comprendre. Pour cette raison, le terme 'entité', associé dans le chapitre 3, à CT est approprié.

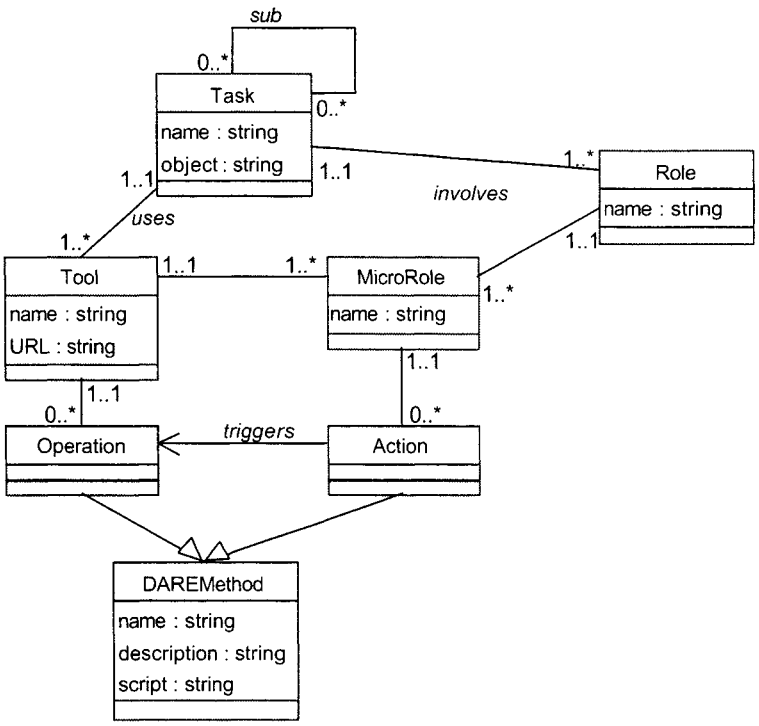


figure 59. Méta-modèle de DARE en MOF

3.2 Les supports du MOF

Dans la section 2.1, un support abstrait d'instanciation et deux supports concrets (XML, Corba) sont abordés. Ils sont en rapport direct à l'opérationnalisation de méta-modèles (sans le niveau M-zéro) évoquée dans le chapitre 3. La principale qualité du MOF est de normaliser cette opération. Les avantages ont été illustrés avec les exemples de transformation de modèles ou de *Data Warehouse*. Au départ (dans la version 1.1 du MOF), l'implémentation d'un méta-modèle consistait en un ensemble d'interfaces IDL et du code associé (qui peut être écrit en Java, C++, SmallTalk, ADA, ...). Les interfaces IDL sont définies à travers un ensemble de règles de projection (*MOF-to-IDL mapping*). L'opérationnalisation peut aussi consister en une DTD. Dans ce dernier cas, l'utilisation est plus statique. Elle est généralement réservée à la persistance.

Néanmoins, la politique de l'OMG évolue vers un attachement moins exclusif à son architecture centrée autour de Corba. Pour cette raison, la version 1.3 du MOF a défini un support d'instanciation abstrait. Il est maintenant possible d'opérationnaliser un méta-modèle défini en MOF dans n'importe quel support : EJB, .NET, mais aussi Java, C++, ... Le MOF se veut encore plus générique : seules subsistent des règles de projection indépendantes de la plate-forme. Pour l'instant seul le support Java a vue une initiative pour "instancier" le support abstrait (JSR40 [JSR 02]).

La description du support abstrait sortirait des objectifs de ce document. C'est principalement le support Corba qui nous intéresse. Il est le seul pour l'instant à proposer un support dynamique distribué. Ce qui correspond à notre problématique de coopération de systèmes. Nous présentons néanmoins le support XML car il constitue la partie la plus utilisée du MOF.

3.3 Le support XML : persistance *

Le but de la projection vers XML est de fournir un format standard pour l'échange de méta-données MOF sous forme de fichiers. Il est difficile d'avoir un format simple et universel pour tout type de méta-données (modèles Merise, UML, programmes Java, description de politique de sécurité,

d'activités coopératives, ...). La structure en deux niveaux du support XML (DTD et document) convient très bien : la DTD représente le méta-modèle, et les documents représentent des modèles. La standardisation consiste, dans un premier temps, à définir la construction d'une DTD à partir d'un méta-modèle décrit en MOF. Ainsi à un méta-modèle MOF donnée, correspondra une unique DTD. Dans un second temps, la standardisation spécifie des contraintes de création de documents XML respectant des DTDs issues des premières règles. L'ensemble de ces règles de projection constitue les spécifications XMI (*XML Metadata Interchange*) [XMI 02].

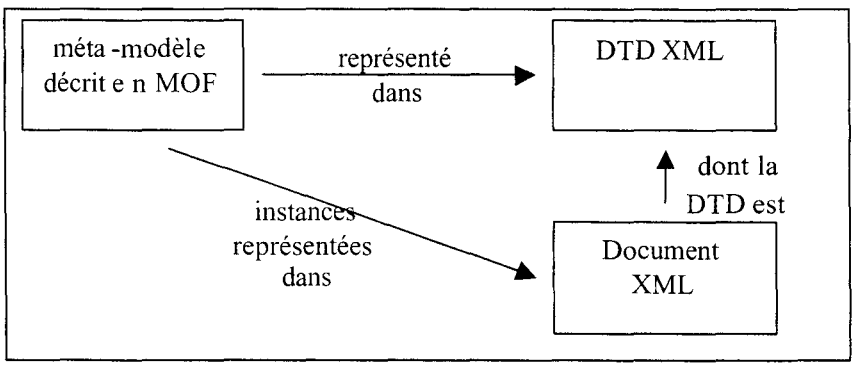


figure 60. Projection vers XML

La figure 60 illustre le modèle de projection et de fonctionnement d'XMI. Un méta-modèle décrit en MOF est traduit en une DTD. Les instances de ce méta-modèle (des modèles) sont représentées dans des documents XML qui respectent cette DTD. Le mécanisme d'instanciation évoqué précédemment est ici réalisé grâce à la relation DTD-document XML.

De manière très générale, la DTD générée pour un méta-modèle donnée, définit une balise XML pour chaque entité et une autre pour chacune de ses caractéristiques. Un document XML qui représente une instance de ce méta-modèle (c'est-à-dire un modèle) en décrit les caractéristiques en utilisant ces balises.

```
<!ELEMENT DARE.Task.name    (#PCDATA|XMI.reference)> . . .
<!ELEMENT DARE.Task.object  (#PCDATA|XMI.reference)> . . .
<!ELEMENT DARE.Task.tools   (#DARE.Tool)> . . .
<!ELEMENT DARE.Task.roles   (#DARE.Role)> . . .
<!ELEMENT DARE.Task.sub     (#DARE.Task)> . . .
<!ELEMENT DARE.Task   ( DARE.Task.name,
                              DARE.Task.object,
                              XMI.Extension*,
                              DARE.Task.tools*,
                              DARE.Task.roles*,
                              DARE.Task.sub*)
?> . . .
```

figure 61. L'entité *Task* dans la DTD "exemple"

La figure 61 montre la déclaration de l'entité *Task* dans une DTD XML générée par les règles XMI. La figure 62 montre une partie du document XML qui correspond à notre modèle *POO_Mod*.

```
<DARE.Task XMI.id='odd.10'>
 <DARE.Task.name>POO_Mod</DARE.Task.name>
 <DARE.Task.object>Module de Programmation Orienté Objet</DARE.Task.object>
 <DARE.Task.sub>
       <DARE.Task XMI.id='odd.11'>
       <DARE.Task.name>LeconA</DARE.Task.name>
       <DARE.Task.object>Bases théoriques</DARE.Task.object>
       <DARE.Task.tools>
             <DARE.Tool XMI.id='odd.12'>
                   <DARE.Tool.name>CoursA
                   </DARE.Tool.name>
                   <DARE.Tool.command>machine1\\\docs\. . .
                   </DARE.Tool.command>
                   . . .
```

figure 62. Document XML pour la tâche POO_Mod

3.4 Le support Corba : représentation mémoire

3.4.1 Fonctionnement général du mode dynamique

La projection d'un méta-modèle MOF vers le langage IDL constitue l'autre réalisation du support abstrait. Le but de cette projection est d'avoir un référentiel dynamique de modèles[55]. Les éléments de ce référentiel sont des objets Corba dont le contenu correspond à la descriptions de modèles, ou plus précisément à des instances d'un méta-modèle MOF (ex : un objet pour représenter le modèle de rôle *Enseignant*). De manière similaire à la projection vers XML, le méta-modèle va servir à spécifier la structure (interfaces IDL) des éléments descripteurs (objets Corba). Ainsi qu'à l'accoutumée en Corba, la création d'objet passe par l'utilisation d'une fabrique (ex : une fabrique de modèles de rôles). La projection vers IDL spécifie aussi la définition d'interfaces IDL pour ces objets fabriques. Le point d'entrée du référentiel est un conteneur de modèles (qui pourrait contenir notre modèle *BD_Mod*). Les principales caractéristiques de ce dernier sont des références sur les fabriques existantes. Chacune de ces fabriques contient une référence vers chaque élément qu'elle a créé (ex : tous les modèles de traitements créés). Le conteneur est lui-même créé à partir d'une fabrique de conteneur (pour permettre une certaine modularité). La figure 63 représente les objets Corba pour notre modèle *Mod_BD*.

[55] Le terme "référentiel" renvoie sur la définition du chapitre 3.

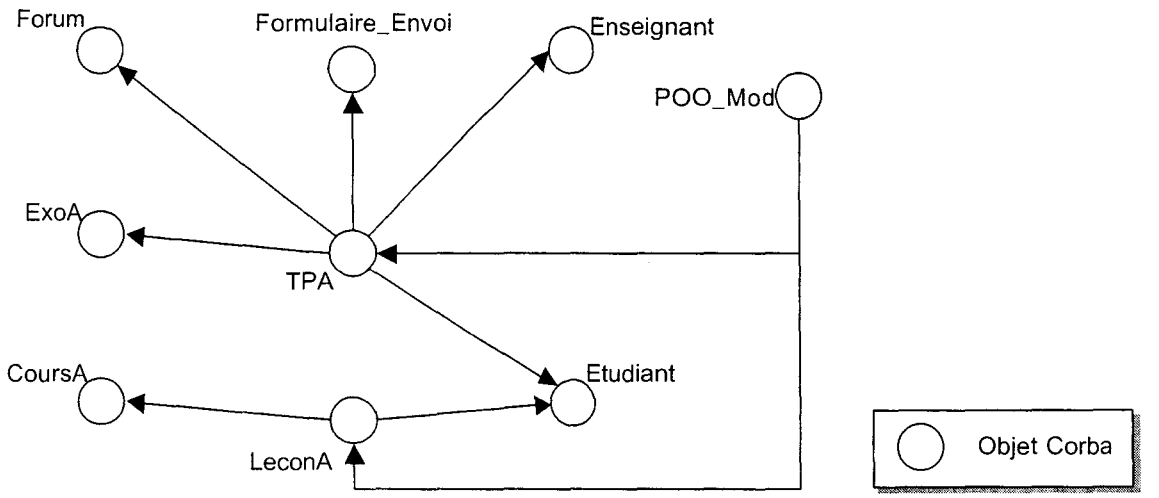


figure 63. Objets Corba/MOF représentant le modèle de tâches d'HLC

La figure 64 représente les liens de création entre les fabriques et les objets descripteurs.

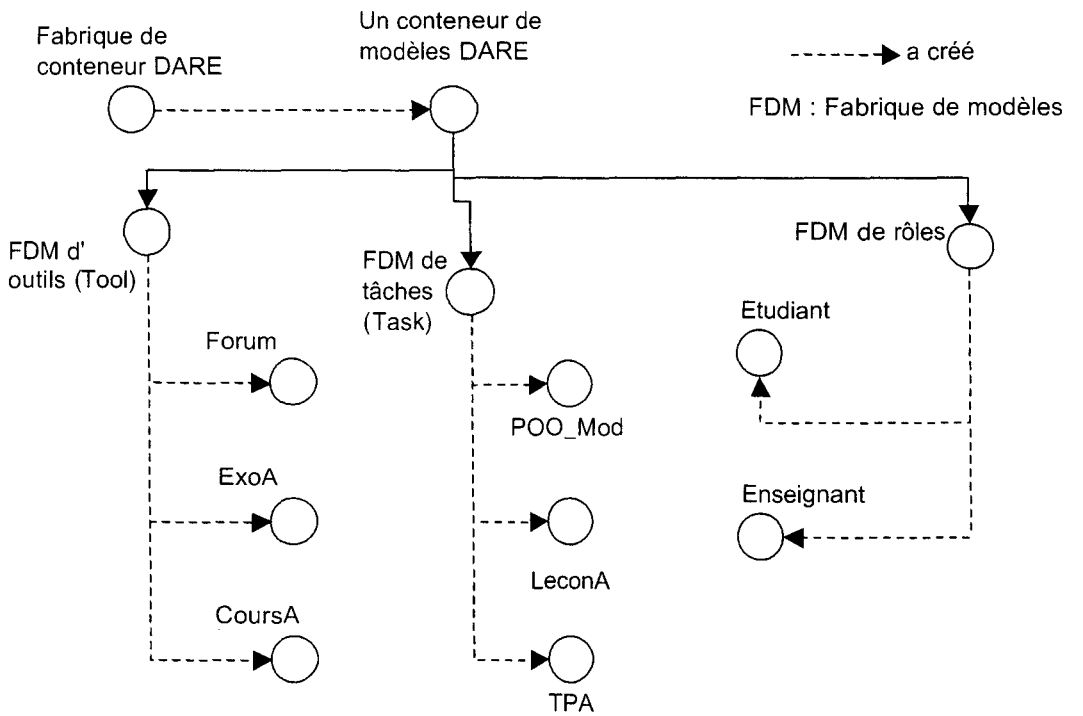


figure 64. Représentation des liens de création

La section suivante montre la structure de chaque objet Corba.

3.4.2 Interfaces IDL *

Les règles de projection vers IDL définissent la structure des futurs objets Corba descripteurs. En voici les principes, à travers le méta-modèle *DARE* :

- Le paquetage *DARE* génère deux interfaces IDL : `DAREPackage` qui est le type IDL des conteneurs de modèles et `DAREPackageFactory`, le type IDL de la fabrique de conteneurs.
- L'entité *Task* génère donne deux interfaces IDL : `Task` qui est le type IDL des objets représentant des tâches et `TaskClass` pour créer les précédents objets (fabrique) et connaître les instances directes ou indirectes de *Task*.
- Chaque attribut ou référence de l'entité génère une ou plusieurs opérations IDL dans l'interface `Task`. L'attribut *name* donne une opération de consultation (`name`) et d'affectation (`set_name`). La référence *sub*, comme elle est multiple, donne en plus des opérations de gestion de liste (`add_sub`, `remove_sub`, ...).
- L'association *TaskTool* génère l'interface IDL `TaskTool` définissant la structure d'un objet Corba régissant tous les liens de type TaskTool : test d'existence de lien entre une tâche et un type d'outil, liste des liens, suppression, modific ation de la liste des types d'outils liées à une tâche. L'association génère aussi une structure IDL `TaskToolLink` (contenant une tâche et un type d'outil) qui est la représentation IDL d'un lien de type *TaskTool*.

La figure 65 montre une partie (simplifiée) des interfaces IDL générées pour le méta-modèle DARE.

```
module DARE {
...
 interface TaskClass :  Reflective::RefObject {

    readonly attribute TaskSet all_of_type_Task; // instances indirectes
    readonly attribute TaskSet all_of_class_Task; // instances directes

    Task create_Task (
        in string name,
        in string object
      );
 };

 interface Task : TaskClass {
     string name ( )  raises...;
     void set_name ( in string new_value )  raises...;
     void unset_name ( )  raises...;


     TaskBag sub ( )  raises...;
     void set_sub ( in TaskBag new_value )  raises...;
     void add_sub ( in Task new_element )  raises...;
     void modify_sub (  in Task new_element ,  in
                                        Task old_element )  raises...;
     void remove_sub ( in TaskBag old_element ) raises...;
  ...
 };
 interface DAREPackage : Reflective::RefPackage {
   readonly attribute TaskClass Task_ref ;
   readonly attribute ToolClass Tool_ref ;
   readonly attribute RoleClass Role_ref ;
```

```
    };

    interface DAREPackageFactory{
      DAREPackage create_DARE_package () raises (Reflective::MofError);
    };
};
```

figure 65. Partie des interfaces IDL générées pour le méta-modèle DARE

3.4.3 Introspection *

Pour accéder à un référentiel Corba-MOF de modèles, il faut connaître la définition du méta-modèle sous-jacent. Il est même logique qu'un programme client "connaisse" la définition du méta-modèle, car il a été créé pour accéder à ce type de référentiel. Néanmoins, certains types de clients ne connaissent pas cette définition. C'est le cas des explorateurs génériques qui peuvent consulter des référentiels issus de n'importe quel méta-modèle. C'est le cas aussi pour certains systèmes qui peuvent supporter une certaine souplesse dans la définition du méta-modèle qu'ils intègrent : sur notre exemple précédent, si l'entité *Role* a un nouvel attribut *Skill* (pour la notion de compétence), ce genre de système (qui intègre ce méta-modèle) permet d'affecter une valeur au champ *Skill* de *Etudiant*. Pour disposer de tels mécanismes dynamiques, il faut implémenter un dispositif qui consulte l'Interface Repository (IR) pour connaître la définition IDL de chaque élément du référentiel pour en déduire le méta-modèle de départ[56].

Pour pallier à ce travail fastidieux, la projection vers IDL définit des mécanismes d'introspection (équivalents à ceux du langage Java). Chaque interface IDL générée hérite de l'interface `RefObject`. Celle-ci définition des opérations qui permettent de connaître le type (MOF) d'une méta-donnée (`ref_meta_object`), d'éditer la valeur des attributs et des références (`ref_value`, `ref_set_value`, `ref_add_value`...) et enfin d'invoquer des opérations (`ref_invoke_operation`).

Nous illustrons ces mécanismes d'introspection à travers la lecture de notre référentiel (contenant le modèle *POO_Mod*) par un explorateur générique (cf. figure 66). Celui-ci a une référence sur le modèle de tâche *POO_Mod*. Pour afficher *POO_Mod*, il a besoin de connaître sa structure. Grâce à l'opération `ref_meta_object` il obtient une référence sur le type de *POO_Mod* : un objet représentant l'entité *Task*. Ce dernier appartient à un référentiel (associé ou non) dont le méta-modèle sous-jacent est le méta-méta-modèle MOF. L'explorateur va deviner qu'il s'agit d'une entité car l'opération `ref_meta_objet` de celui-ci renvoie sur l'objet représentant la méta-entité *Class*. Avec l'invocation de l'opération `contents`[57] sur l'objet *Task*, l'explorateur va disposer des objets représentant respectivement les attributs, les références et les opérations de l'entité *Task*. Lorsque l'explorateur présentera l'objet *POO_Mod* dans son interface, l'affichage de celui-ci correspondra à toutes les valeurs correspondantes à chaque caractéristique récoltée précédemment. Pour connaître, par exemple, la valeur d'*object*, l'explorateur invoquera sur celui-ci l'opération `ref_value` avec comme paramètre l'objet représentant l'attribut *object*.

[56] Toutes les caractéristiques de chaque entité ne seront pas retrouvés.

[57] La référence *contents* de Class contient la définition d'une classe (attributs, référence, opérations, classes incluses, ...). Cette opérations fait partie de l'interface IDL *Class* issue de la projection du méta-méta-modèle MOF (lui-même) vers IDL. Cette opération est donc connue.

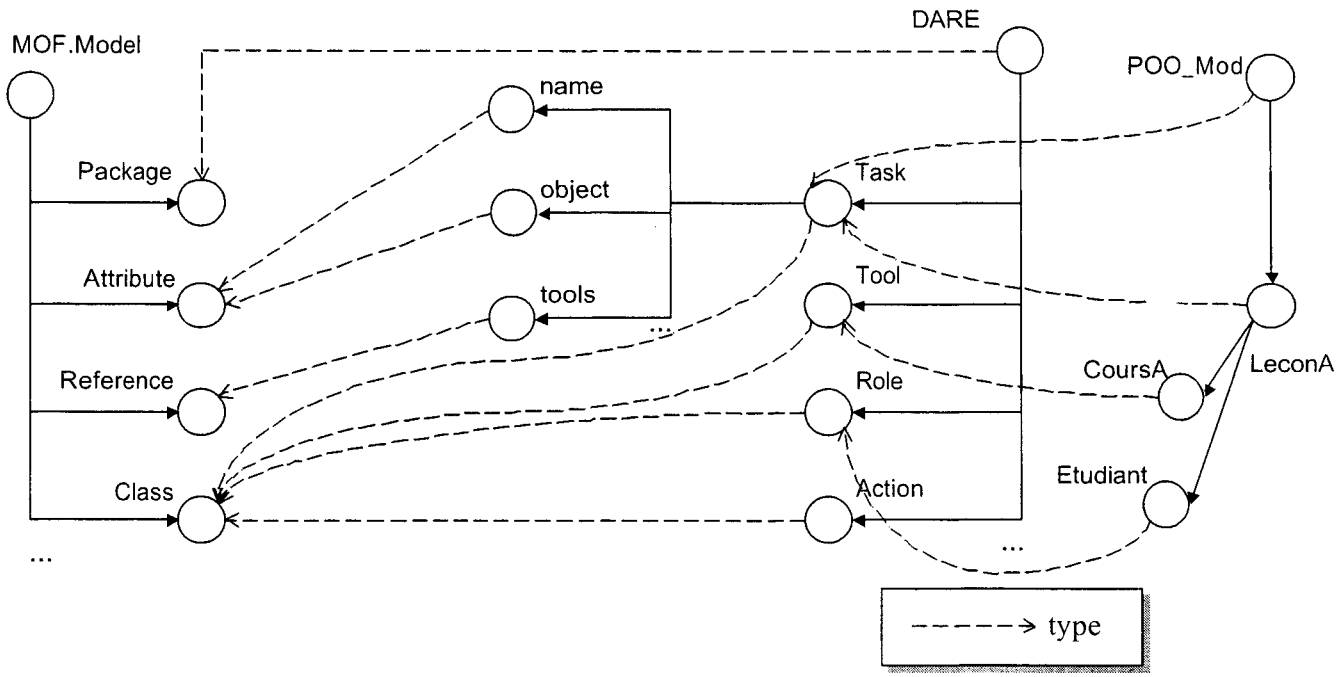


figure 66. Objets Corba représentant des éléments du modèle de tâche d'HLC, des entités de DARE et des méta-entités MOF

Le développement d'un lecteur générique ou de systèmes avec des capacités de modélisation plus flexibles est donc facilité par le mécanisme d'introspection du MOF.

3.5 Les outils MOF existants

La figure 67 montre le fonctionnement (théorique) d'un outil MOF c'est-à-dire d'une implémentation complète des spécifications du MOF. À partir d'une description graphique (en UML) ou textuelle (en MODL) d'un méta-modèle, l'outil peut générer la DTD correspondante (opération 2 sur la figure) ou les interfaces IDL correspondantes et leur implémentation, généralement des classes Java (opération 3). Dans le dernier cas, le concepteur doit compiler les interfaces IDL, et compiler les classes Java qu'il a éventuellement modifiées (opération 4). Ces classes implémentent un mécanisme de persistance XMI. Ainsi, à l'exécution, il est possible de sauvegarder au format XMI un modèle de données que représentent un ensemble d'objets Corba ou de créer des objets Corba représentant un modèle de données décrit dans un fichier XMI (opération 5).

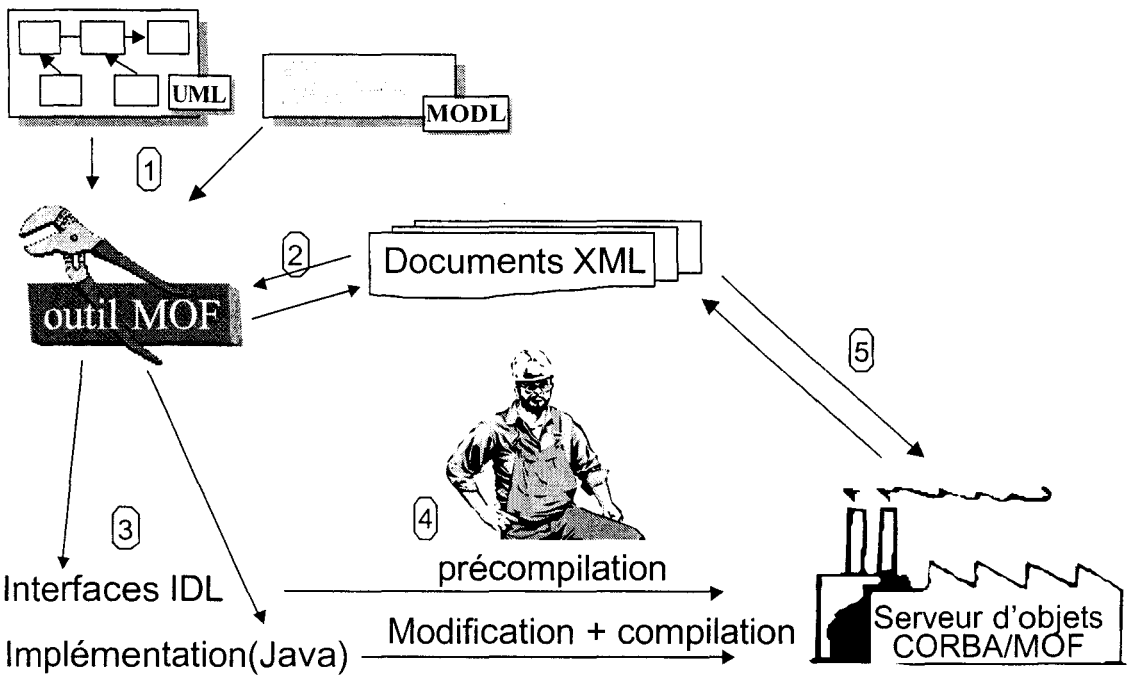
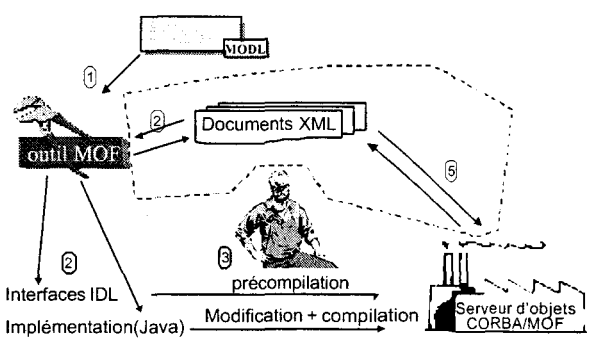


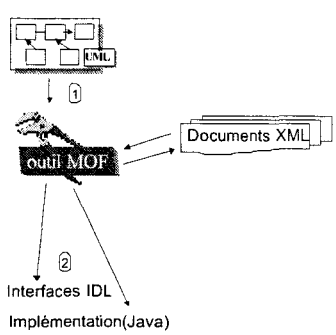
figure 67. Fonctionnement d'un outil MOF

À l'heure actuelle, nous avons recensé six outils MOF. Aucun d'entre eux ne proposent les cinq types dopérations précédentes. La présentation de ces six outils commence par les outils les plus fournis.

dMOF [DMO 02] : Cet outil a été développé par un des deux principaux auteurs du MOF (le laboratoire DSTC). dMOF devrait fournir toutes les fonctionnalités précitées mais à l'heure actuelle, il ne fournit que les opérations 2 et 3. La description du méta-modèle se fait uniquement en MODL. La génération de la DTD correspondant à un méta-modèle est la dernière fonctionnalité proposée.



M3J [M3J 02] : Le développement d'M3J s'inscrit dans les travaux de thèse de X. Blanc [BLA 01]. M3J peut générer pour un méta-modèle décrit en UML (au sein de l'outil) la DTD ou les interfaces IDL correspondantes (auxquelles est associés une implémentation Java).



Metadata Repository [MDR 02] : MDR est développé par Sun dans l'un de ses outils de conception d'application. À partir d'une description XMI d'un méta-modèle, MDR génère les classes Java correspondantes. Il ne s'agit pas ici d'utiliser le support Corba mais le support Java (initiative JSR 40).

Rational Rose (plug-in MOF/XMI) : le plug-in MOF/XMI de l'outil UML Rational Rose permet de définir un méta-modèle MOF et de le sauvegarder au format XMI.

Universal REPository : UREP est un outil de conception d'applications développé par l'autre auteur principal du MOF (Unisys). UREP a comme particularité de permettre la création d'un référentiel d'objets (le langage de définition est propriétaire) lesquels sont accessibles à partir d'un grand nombre de langages de programmation de manière transparente. Le référentiel devait, à la création du MOF, être compatible avec le standard. Ceci n'a jamais été le cas. Unisys a, depuis l'année 2001, stoppé la vente d'UREP.

XMI Toolkit [XTK 02] : l'équipe alphaWorks d'IBM propose l'outil XMI Toolkit. Il permet de faire la conversion d'un modèle UML en une DTD MOF/XMI (le modèle est alors considéré comme un méta-modèle MOF) ou en un modèle de classes Java défini dans un document XML référençant la DTD XMI de Java.

Les spécifications du MOF définissent un méta-méta-modèle, fort proche d'UML, afin de décrire des méta-modèles. Elles définissent aussi deux supports d'instanciation, Corba et XML (spécifications XMI), dont nous avons décrit ici les principes. Une demi-douzaine d'implémentations des spécifications du MOF existe. Si elles proposent déjà un certain nombre de fonctionnalités, celles-ci devraient par la suite augmenter (principalement pour l'outil dMOF).

4 Le MOF et le niveau M-zéro

L'objectif du MOF est de gérer les éléments de niveau M1, c'est-à-dire des modèles. Les supports d'instanciation pour les méta-modèles (Corba, XML) ne proposent pas de double instanciation : la gestion des éléments M0 n'est pas offerte. L'implémentation de l'instanciation des concepts d'instance et des liens avec les éléments M1 est donc à la charge des concepteurs utilisant des outils MOF. Cependant il est toujours possible d'utiliser le MOF pour le niveau M0 : faire apparaître les concepts d'instance dans le méta-modèle. Mais ils n'y sont que de simples entités (le caractère spécifique du concept d'instance n'est pas présent). Cette solution n'est cependant pas à écartée et se révèle même judicieuse pour deux raisons : la génération d'une partie de l'implémentation et le gain de temps dans le cadre de l'interopérabilité. Ces deux avantages sont détaillés dans la section 4.1. La section suivante indique comment définir des concepts d'instance dans un méta-modèle MOF et quels sont les mécanismes additionnels (spécifiques au niveau M-zéro) à implémenter. Devant le travail conséquent et répétitif que représente l'implémentation, nous proposons la modification des spécifications du MOF pour que celui-ci intègre la notion de concept d'instance et, par la suite, que les outils MOF génèrent la "couche" M-zéro (section 4.3). Cette partie se termine sur l'impact de ces extensions M-zéro sur CAST et son application au MOF.

4.1 Avantages à utiliser le MOF pour le niveau M0

Cette implémentation reprend les aspects évoqués dans le chapitre 3. CAST étant appliquée au MOF, les adaptateurs suivent la structure définie par le MOF. Pour que notre solution fonctionne avec les éléments M0, il faut que les concepteurs utilisent le MOF pour gérer ce type d'éléments. Si un équivalent du MOF pour les éléments M0 existait, nous aurions appliqué CAST sur celle-ci.

4.1.1 Accélération de développement

L'utilisation d'un langage de modélisation plus évolué et d'une projection normalisée peut aussi apporter un gain de temps dans la conception même du système. Un langage de modélisation plus

évolué signifie précisément que l'expression d'un phénomène est moins coûteuse. UML est plus évolué (i.e plus fourni) que les sNet (réseaux sémantiques)[58] : par exemple, la description d'une association en UML est plus compact (en sNet, il faut créer un nœud pour la cardinalité, un nœud pour exprimer le nom des rôles, l'existence des références, ...). Un seul concept en UML peut signifier une construction de plusieurs concepts en sNet.

Cette différence est très présente entre UML et les langages de programmation. Les langages de programmation ont un grand pouvoir d'expression (grâce aux instructions qu'ils proposent) mais la présence de concepts évolués est rare. Au lieu d'utiliser des formalismes de modélisation comme UML, bon nombre de concepteurs développent encore des applications et des systèmes directement d'un langage de programmation. Pourtant il est très judicieux d'utiliser des langages de modélisation évolués comme UML pour la description d'applications ou de systèmes, car cette description se révèle plus rapide, plus compacte. Il est bien sûr nécessaire que ces formalismes évolués soient associés à des générateurs de code d'implémentation. C'est le cas d'UML avec des outils comme Rationale Rose, Objecteering, ArgoUML... C'est aussi le cas du MOF et des outils dMOF et M3J. L'avantage du MOF est que les règles de génération sont normalisées. Ce qui n'est pas le cas d'UML ou d'autres formalismes comme Merise.

Le MOF apporte donc un gain de temps pour implémenter la couche M0 (description compacte) avec l'avantage d'avoir des règles de génération de "code" normalisées.

4.1.2 Accélération de la mise en correspondance

L'utilisation du langage de modélisation plus évolué n'apporte pas qu'un gain de temps pour le développement. L'accélération intervient aussi dans le cadre de l'interopérabilité. Faire des liens de correspondance entre deux modèles UML est plus rapide que la même opération entre les programmes Java correspondants. Un seul lien d'équivalence entre deux références UML correspond à des liens entre les méthodes de consultation (get), d'affectation (set, add, modify, remove) et entre les méthodes de gestion des liens des classes d'associations (si elles existent).

L'emploi du MOF pour implémenter le niveau M0 est plus "intelligente" que la simple utilisation du langage IDL. Un apport sémantique résulte de l'utilisation du MOF. Pour notre adaptation des niveaux M0 de chaque système, il sera plus rapide d'établir des liens entre des associations, des références, des attributs MOF qu'entre des opérations IDL.

4.2 Les concepts d'instances dans les méta-modèles MOF

Implémenter la gestion des éléments M0 avec le MOF revient à définir les concepts d'instance dans le méta-modèle. Cette définition suit les règles générales énoncées dans le chapitre 3. Dans cette section, nous détaillons l'implémentation des CIs qui renferme des mécanismes complexes.

4.2.1 La définition

La définition d'un CI est constituée d'un lien vers le ou les concepts de type associé(s), d'une partie des liens définis par le ou les CT - avec une instanciation des types de valeurs - et de caractéristiques propres au CI - ex : pour un workflow, son état (en attente, en cours, terminé). La figure 68 montre la définition des CI associés à *Task* de DARE en MOF. Les caractéristiques en gras de *Activity* montre ce qui est lui propre, c'est-à-dire non déductible de *Task*. L'opération *create* de *Task* est l'opération d'instanciation de *Task* qui permet à une tâche de créer des instances. Nous proposerons dans la section suivante (4.3) des extensions au MOF pour faciliter ce type de définition.

[58] Ce qui est une conséquence logique d'un des objectifs des sNet : avoir un noyau minimal. Avoir des descriptions compactes ne font pas partie des objectifs des sNets.

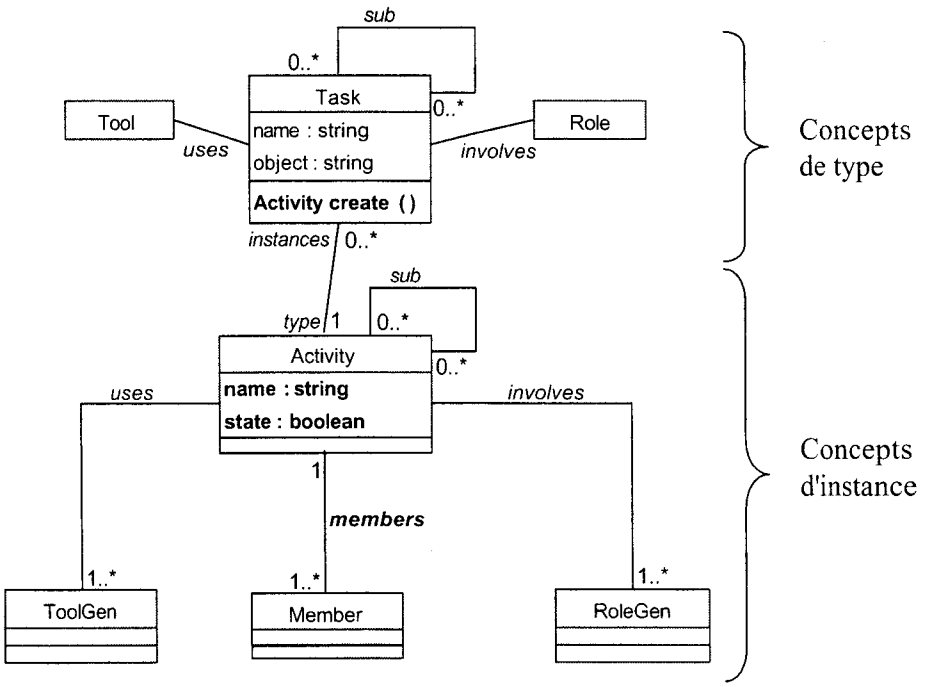


figure 68. Définition du concept d'instance Activity en MOF

4.2.2 L'implémentation

Le comportement d'un élément M0 est une spécialisation du comportement d'un élément M1 : il reprend les mêmes mécanismes au sein des desquels il ajoute de nouvelles opérations. Ces ajouts sont l'objet des cinq paragraphes suivants.

4.2.2.1 Typage des caractéristiques issues des CT

Les CI "héritent" de caractéristiques de leur(s) CT associé(s). Le type de ces caractéristiques héritées est le même mais à une instanciation près. Ainsi que le montre la figure 68, une tâche référence, à travers *involves*, des types de rôles alors qu'une activité (instance d'une tâche) référence, à travers *involves*, des rôles. La vérification de typage qui a lieu dans les opérations d'affectation (`set`, `add`, `modify`) d'*involves* pour une activité consiste d'abord à vérifier que le ou les éléments passés en paramètre soi(en)t de(s) rôles, c'est-à-dire une(des) instance(s) de *RoleGen*. Nous appelons *RoleGen* le *type primaire* (de la référence). Une deuxième vérification consiste à vérifier si ces rôles ont comme type de définition des types de rôle référencés par le type de définition de l'activité. Nous appelons ces types de rôles les *types secondaires*. Par exemple, une activité *aTPA* dont le type de définition est *TPA* ne peut référencer que des rôles dont le type de définition est *Enseignant* ou *Étudiant*. Le type primaire de la référence *involves* de *aTPA* est *RoleGen* et les types secondaires sont *Enseignant* et *Étudiant*. Cette **vérification de typage additionnelle** est le premier mécanisme à implémenter pour la gestion des éléments M0.

Note : Pour simplifier nos propos, la notation aTPA:Activity[TPA] indique que aTPA est une instance d'Activity (type réel) et que son type de définition est TPA. Une variation de cette notation est "aTPA est :Activity[TPA]". S'il y a plusieurs types de définition, ils sont séparés par des ";".

4.2.2.2 Cardinalité des caractéristiques des éléments M1

Chaque élément M1 doit associer une cardinalité à chaque élément qu'il référence (dans un attribut ou dans une référence). Une activité *:Activity[TPA]* peut-elle référencer plusieurs instances d'*Étudiant* ou est-elle limitée à une seule ? Cette question se traduit par "y a-t-il plusieurs ou une seule personne qui joue le rôle d'étudiant ?". La question peut se poser aussi pour *Enseignant*.

Cette particularité implique l'implémentation de deux mécanismes :

1. Ajouter, dans les interfaces IDL générées par l'outil MOF, des opérations IDL (et les implémenter) pour associer des cardinalités aux valeurs que contiennent les caractéristiques qui sont répercutées sur les concepts d'instance.
2. Implémenter une vérification de la cardinalité dans les opérations d'édition des caractéristiques des CI héritées des CT. Par exemple, si la cardinalité associée à *Enseignant* est 1, il n'est pas possible d'ajouter un deuxième rôle *Enseignant* à jouer s'il y en a déjà un de référencé.

Nous appelons **cardinalité secondaire**, la cardinalité associée aux éléments M1.

4.2.2.3 Généralisation/spécialisation entre éléments M1

L'héritage est un mécanisme très pratique dans le cadre de la réutilisation. Avec une gestion du niveau M0 par le MOF, la possibilité de définir un lien d'héritage dans un méta-modèle entre concepts de type devient très utile. Par exemple, un simple lien d'héritage vers une tâche permet à un nouveau type d'activité de ré-utiliser le comportement que la tâche définit.

Permettre à des instances d'un concept de type de se spécialiser (entre eux) nécessite deux opérations :

1. définir pour le CT une association sur lui-même.
2. modifier l'implémentation des opérations d'édition des caractéristiques où apparaît le CI associé au CT.

L'implémentation vise précisément à intégrer deux nouveaux mécanismes. Le premier consiste à étendre l'ensemble des types secondaires pour une caractéristique à ceux définis par les types "pères". Par exemple, nous ajoutons à l'entité *Task* une association d'héritage. Si la tâche *TPA* hérite de *coursA*, une activité *TPA* peut référencer, comme outils, un *exoA*, un forum, un formulaire d'envoie et *aussi* un coursA. De plus, la tâche *TPA* n'a plus à référencer le rôle *Étudiant* car il est déjà défini dans *CoursA*.

Le polymorphisme constitue le second mécanisme à implémenter. Prenons un exemple où l'entité *Role* dispose elle-aussi d'une association "héritage". Le rôle *MaîtreDeConférences* hérite d'*Enseignant*. Une activité *TPA* peut accepter comme rôle, un enseignant mais *aussi* un maître de conférences, car celui-ci est aussi un enseignant.

4.2.2.4 Liaison caractérisante

Certains concepts de type peuvent avoir, dans leurs objectifs, de caractériser des éléments. Le concept de Rôle définit des types de rôle, c'est-à-dire des éléments indépendants, qui existent en tant que tel. Mais il permet aussi de caractériser une tâche. Il doit être possible de demander à un *TPA* quels sont les étudiants qui y participent. Le lien inverse (tâches dans lesquelles intervient un type de rôle) est moins caractérisant : quels sont les *TPAs* où est joué un rôle *Étudiant* particulier ? C'est une question intéressante mais qui ne nécessite peut-être pas son propre point d'entrée.

Les points d'accès d'un élément M0 sont généralement définis par les caractéristiques de son concept d'instance (dont ceux sont hérités des concepts de type associés). Par exemple, si *tpa_Alain* est *:Activity[TPA]*, ses points d'accès sont *tpa_alain.roles*, *tpa_alain.state*, *tpa_state.name*, *tpa_sub*, etc. Si on définit la liaison *roles* comme caractérisante, le nombre de points d'accès des instances d'Activity va augmenter : *tpa_Alain* a comme les points d'accès supplémentaires : *tpa_Alain.etudiant* et *tpa_Alain.enseignant*. Néanmoins, l'ajout de points d'entrées est une opération lourde. Dans le cas de MOF-Corba, il faut ajouter des opérations à l'interface IDL correspondante au concept d'instance. Si on souhaite avoir les points d'entrée *etudiant* et *enseignant* pour les instances de *TPA* il faut créer une interface IDL spécifique à *TPA* (héritant de `Activity`). Dans ce cas, le référentiel crée à chaque nouvel élément M1, une interface IDL correspondante. Les applications "clientes" du référentiel doivent être capables de lire des activités sans connaître au préalable leur interface IDL. Corba

propose certes les mécanismes nécessaires à ce scénario (le *Dynamic Skeleton Interface* et l'*Interface Repository*) mais ils restent actuellement lourds à l'usage.

La création de points d'entrées *secondaires* est préférable. Par "secondaires", nous entendons des opérations comme `get_roles(String name)` qui s'utilisent de la manière suivante `get_roles("enseignant")`. Ces opérations plus génériques sont plus légères à gérer pour le référentiel et l'implémentation des applications clientes s'en trouve allégée.

Qu'elles soient directement répercutées sur les interfaces IDL ou indirectement à travers des opérations IDL génériques, les liaisons caractérisantes nécessitent l'implémentation de mécanismes de filtrage.

4.2.2.5 Intercession

Le support Corba du MOF est destiné à créer, lire et modifier des éléments M1[59]. L'intégration du niveau M0 ne doit pas limiter les opérations possibles sur les éléments M1. Les éléments M0 qui sont typés par des éléments M1 doivent donc avoir la capacité d'effectuer des répercussions lors de la modification de leur(s) type(s). Par exemple, retirer la référence sur *Enseignant* de *TPA* supprime les rôles *enseignant* joués dans les *Activity[TPA]* en cours : les personnes présentes dans les *TPAs* en tant que professeur en sont éjectés. Ce phénomène s'apparente à l'intercession des langages réflexifs (comme Smalltalk). Ce type de mécanismes est assez lourd à implémenter :

- Le parcours de toutes les instances et la modification de chacune d'elles sont le type de traitements qui s'ajoute à chaque opération d'édition d'éléments M1. Le changement de la cardinalité d'une caractéristique, le type, le nom, ... entraîne des répercussions.
- La structure interne associée au concept d'instance doit être souple. Parce qu'une sous-caractéristique (ex : enseignant) peut changer de type, de cardinalité, ... il doit être aisé d'effectuer la répercussion.
- L'héritage vient complexifier les répercussions. Si *tpa_Alain* référence un maître de conférences et que le lien d'héritage entre *Enseignant* et *MaitreDeConferences* est supprimé, la précédente référence de *tpa_Alain* (vers un maître de conférences) doit être supprimée. Si *TPA* héritait de *CoursA* (qui définissait la présence d'étudiants), et que ce lien est supprimé, les étudiants référencés par *tpa_Alain* ne le sont plus.

4.3 Extensions du MOF pour le niveau M0

La gestion d'un niveau M0 n'est pas un luxe, mais une pièce maîtresse des supports informatiques au système d'information. Son implémentation est conséquente. Elle peut néanmoins être générée par un outil. Pour cette raison, il nous paraît intéressant d'intégrer la notion de concept d'instance dans le MOF afin que les outils MOF-Corba génère les interfaces IDL et le code associé pour la gestion du niveau M0. Cette intégration se fait au niveau du méta-méta-modèle MOF, de la notation graphique et des règles de projection IDL.

Les extensions que nous apportons au MOF permettent au concepteur d'avoir une vue plus claire des trois "niveaux". Elles nous permettent aussi d'exprimer les liens forts entre le niveau M1 et le niveau M-zéro, ce qui nous servira dans l'application de CAST sur le support MOF.

[59] La modification n'est pas tout le temps permise. Les propriétés *isChangeable* et *isFrozen* d'un attribut ou d'une référence conditionne la génération des opérations *set_*, *add_*, *remove_*, *modify_*.

4.3.1 Modification du méta-méta-modèle MOF

La figure 69 montre les ajouts que nous proposons pour le MOF afin d'intégrer le concept d'instance. La figure ne montre que les ajouts.

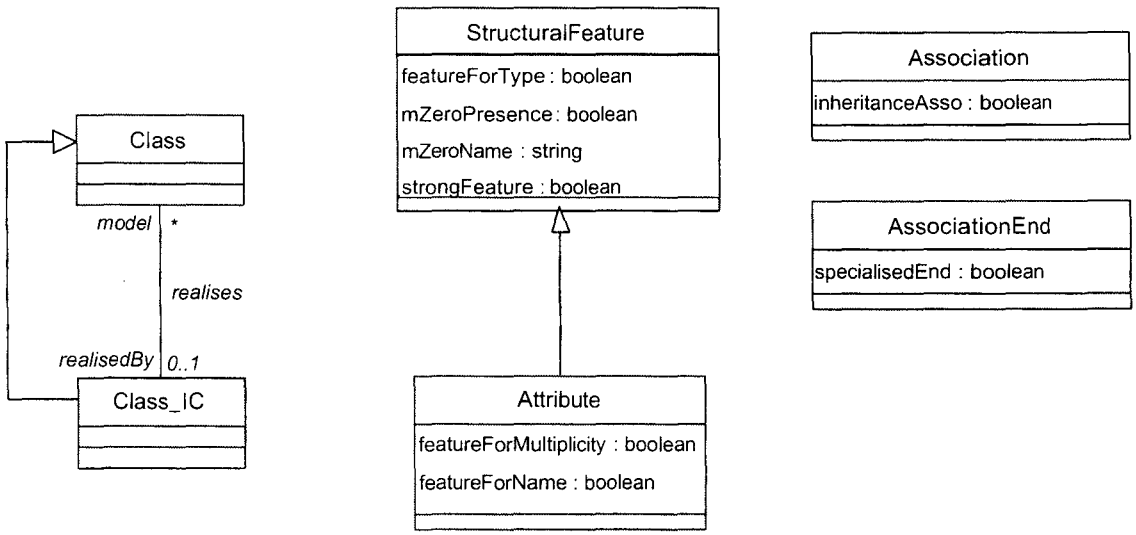


figure 69. Intégration du Concept d'instance dans le méta-méta-modèle MOF

4.3.1.1 Class_IC : Concept d'instance

La méta-entité *Class_IC* définit ce qu'est un concept d'instance. Les concepts de type, c'est-à-dire les entités, sont quant à eux, toujours définis par la méta-entité *Class*.

Class_IC hérite de *Class*, c'est-à-dire qu'il est possible de définir des références, des attributs et des opérations. Il est aussi possible de lui associer des classes (et donc aussi des classes_IC). Une caractéristique définie dans une *classe_ic* apparaît pour chacun des instances de celle-ci. Par exemple, la définition de l'attribut *state* dans *Activity*, indique que toutes les activités, quelque soit leur(s) type(s) de définition, ont une propriété *state*.

Chaque concept d'instance est lié avec un ou plusieurs concepts de type qu'il réalise. Cette propriété est matérialisée par l'association *realises* : un concept de type est le modèle d'un concept d'instance (*model*) - un CI réalise un CT (*realisedBy*). Une *Class* peut être réalisée par un CI. Dans ce cas, la classe est obligatoirement un CT. Un CI peut être la réalisation de plusieurs CT.

4.3.1.2 Le lien d'héritage

L'existence de liens d'héritage entre éléments M1 est d'abord une association "fermée" sur le type de ces éléments[60]. Pour désigner cette association, nous ajoutons un "flag" (i.e. un attribut) *inheritanceAsso* à la méta-entité *Association*. Si nous créons une association *specialisation* sur *Task*, il nous suffit de mettre *inheritanceAsso* à vrai pour que des mécanismes d'héritage soit générés et attachés à cette association et à ses deux références (si elles existent). Le flag *specialisedEnd* d'*AssociationEnd* indique qu'elle est l'extrémité de l'association qui constitue la spécialisation.

Au départ, nous avions simplement défini un flag *featureForInheritance* dans la méta-entité *Feature* dont hérite *Attribute* et *Reference*, c'est-à-dire les caractéristiques structurelles. Il était possible grâce à lui, d'adjoindre les mécanismes d'héritage à un possible attribut *specialise* de *Task*. Néanmoins, nous avons jugé que l'héritage implique des éléments généralisés ET des éléments spécialisés. C'est

[60] Nous ne gérons pas ici un héritage entre éléments de types différents (sauf s'ils sont liés eux-mêmes par héritage).

une réelle liaison et pas un simple référencement (un attribut ne convient pas). De plus vis-à-vis de l'intercession, il est important pour un élément M1 de connaître ses "spécialisateurs" : lorsqu'une caractéristique est modifiée, il faut en répercuter les modifications sur les instances, c'est-à-dire les instances du type définissant la caractéristique et les instances des sous-types. Pour ces raisons le flag *featureForInheritance* fut supprimé au profit de *inheritanceAsso* et *specialisedEnd.*

4.3.1.3 Répercussion d'une caractéristique sur le concept d'instance

Certaines caractéristiques structurelles d'un CT se répercutent sur son CI. Pour indiquer quelles sont les caractéristiques qui seront présentes sur les deux niveaux, nous définissons le flag *mZeroPresence* sur *StructuralFeature*[61]. Ainsi il est possible d'indiquer si une référence ou un attribut est répercuté sur le niveau M0. Les opérations[62] ne peuvent être répercutées directement. Elles contiennent un comportement/traitement qui ne peut être appliqué sur deux niveaux. La répercussion peut être faite mais il faut instancier les types des paramètres qui le peuvent[63] et le concepteur doit changer l'implémentation de l'opération. De plus, les opérations étant assez rares dans les méta-modèles, nous n'avons pas jugé nécessaire la présence de moyens pour indiquer la répercussion d'opération.

Le flag *mZeroName* d'une caractéristique indique le nom de la caractéristique répercutée.

4.3.1.4 Les liaisons caractérisantes

Certaines références ou attributs définis dans des CT ont un pouvoir fortement caractérisant sur les CI. Nous avons appelé celles-ci, les liaisons caractérisantes. Encore une fois nous avons ajouté un flag (*strongFeature*) à *StructuralFeature* qui permet d'indiquer si les valeurs de la caractéristique d'un élément M1 produisent des points d'entrée secondaires sur les instances de ce dernier.

Chaque valeur référencée à travers une telle caractéristique va définir un point d'entrée secondaire sur les instances. Un point d'entrée est principalement caractérisé par un nom, un type de valeur et une multiplicité[64]. Ces trois informations sont déduites de la précédente valeur référencée. Le type, et le nom sont respectivement la valeur et son nom. Pour savoir quelle propriété de la valeur correspond au nom, nous avons défini un flag *featureForName*. Il faut mettre ce flag à vrai sur la caractéristique (définie dans le type de la valeur) qui correspond au nom.

Pour le type, il ne s'agit pas tout le temps de la valeur elle-même. C'est le cas des concept de Class et d'Attribut du paradigme objet. Par exemple, l'attribut *title* de la classe Java *Frame* définit une propriété (sur les instances de *Frame*) dont le type n'est pas l'attribut *title* mais *String*. Pour ce type de situations, nous avons créé le flag *featureForType.* Il désigne quelle est la propriété qui correspond au type. Sur l'exemple précédent, ce flag est à vrai sur la référence *type* de *Java.Attribute*.

Enfin pour la multiplicité, il faut obligatoirement se référer à une propriété de la valeur comme celles que l'on trouve souvent dans la définition d'attribut. C'est l'objectif de *featureForMultiplicity* : il indique la propriété "multiplicité". Néanmoins nous ne pouvons pas exiger ce genre de propriété à chaque fois. Par exemple, le concept *Role*, type des valeurs de la référence *involves*, ne définit pas une telle caractéristique. Dans notre implémentation des extensions M-zéro (cf. chapitre 6), nous associons automatiquement une multiplicité à chaque valeur d'une propriété d'un élément M1. Pour une liaison caractérisante, si aucune caractéristique du type de valeur n'est une *featureForMultiplicity*, c'est la multiplicité automatique qui est utilisée.

---

[61] *Attribute* et *Reference* héritent de *StructuralFeature*.

[62] *Operation* héritant de *BehavioralFeature*.

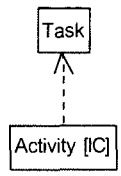
[63] Un type de base (entier, chaîne de caractères,...) comme type de valeur d'une caractéristique

[64] D'autres éléments de définition peuvent bien sûr intervenir (ex : *readonly*), mais les trois précédents constituent pour nous un minimum suffisant dans le contexte du MOF et des langages objet à typage fort.

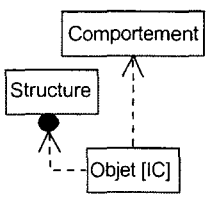
4.3.2 Notation graphique

Chaque extension du méta-méta-modèle MOF doit pouvoir être utilisable dans la définition d'un méta-modèle. Pour cela nous avons aussi inséré de nouveaux artefacts dans la notation graphique. La figure 70 montre l'utilisation de ces artefacts sur les concepts *Task* et *Activity*. Nous expliquons chacun ajout dans le formalisme UML/MOF.

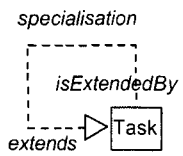
L'expression [IC] (*Instance Concept*) qui suit le nom d'une classe MOF indique qu'il s'agit d'une *Class_IC*, c'est-à-dire un concept d'instance. Celui-ci est relié à son ou ses concepts de type par le lien *realises* d'UML (une flèche en pointillé). En UML *realises* lie une classe à une interface : une classe remplit le contrat défini par l'interface (i.e. un ensemble d'opérations à proposer). Ici nous réutilisons donc cet artefact avec une signification différente.



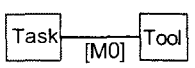
Une instance d'un CI peut être définie à travers plusieurs instances issues de CT différents (comme pour le système *Endeavors* [KAM 00]). Pour indiquer où l'opération de création/instanciation apparaît, nous avons ajouté un rond plein. Il intervient seulement lorsqu'un CI est associé à plusieurs CT. Insérer un rond plein signifie les instances du CT disposeront de l'opération de création. Cette opération prendra en paramètre une instance pour chaque autre CT.



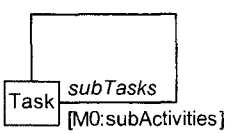
Pour désigner une association d'héritage (*inheritanceAsso* à vrai), nous avons utilisé le lien d'héritage habituel (flèche avec un triangle blanc) mixé le lien *realises* pour indiquer qu'il sera réalisé sur les concepts d'instance. Le rôle indiqué à côté du triangle signale le "spécialisateur" (la figure ci-droite montre une autre possibilité pour le nom des rôles).



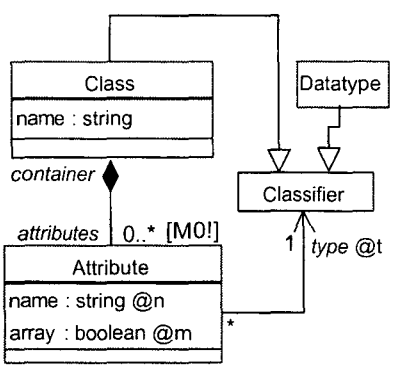
L'expression [M0] à droite d'un rôle ou d'un attribut (pour un concept de type) implique une répercussion de la référence associée ou de l'attribut sur le concept d'instance associé (*mZeroPresence* à vrai). Un [M0] au milieu d'une association correspond au deux rôles sont notées [M0].



Si le nom de la caractéristique répercuté est différent (*mZeroName*), il est indiqué entre les crochets séparé du *M0* par un ":". L'exemple ci-droite est fictif.



Les liaisons caractérisantes ne sont qu'une sous-catégorie des caractéristiques se répercutant sur le niveau M0. Ces dernières étant annotées [M0], nous avons juste ajouté le '!' au sein des crochets pour signaler qu'il s'agit d'une liaison caractérisante (*strongFeature* à vrai). Pour désigner les caractéristiques de nommage (*featureForName* à vrai), de typage (*featureForType* à vrai) et de multiplicité (*featureForMultiplicity* à vrai), il faut respectivement utiliser les expressions @n, @t et @m. Pour la multiplicité, les types acceptés sont *MultiplicityType* (structure défini dans le méta-méta-modèle MOF, et les types de base *Boolean*, *Long* et *Ulong*.



La figure 70, par rapport à la figure 68 (définition des CI sans les extensions M-zéro) qui est ré-intégrée dans la figure 70, est plus allégée (et donc plus rapide à écrire) :

- les références *uses*, *involves* et *sub* n'apparaissent qu'au niveau des concepts de type,
- *Task* n'a pas d'opération *create*,

- Le lien *realises* est plus léger que l'association *realises*.

La figure 70 est aussi beaucoup plus significative que la figure 68 : les deux associations *involves* n'étaient pas forcément liés sur la figure 68, l'association *realises* n'indiquait pas nécessairement un lien d'instanciation ou de typage, ...

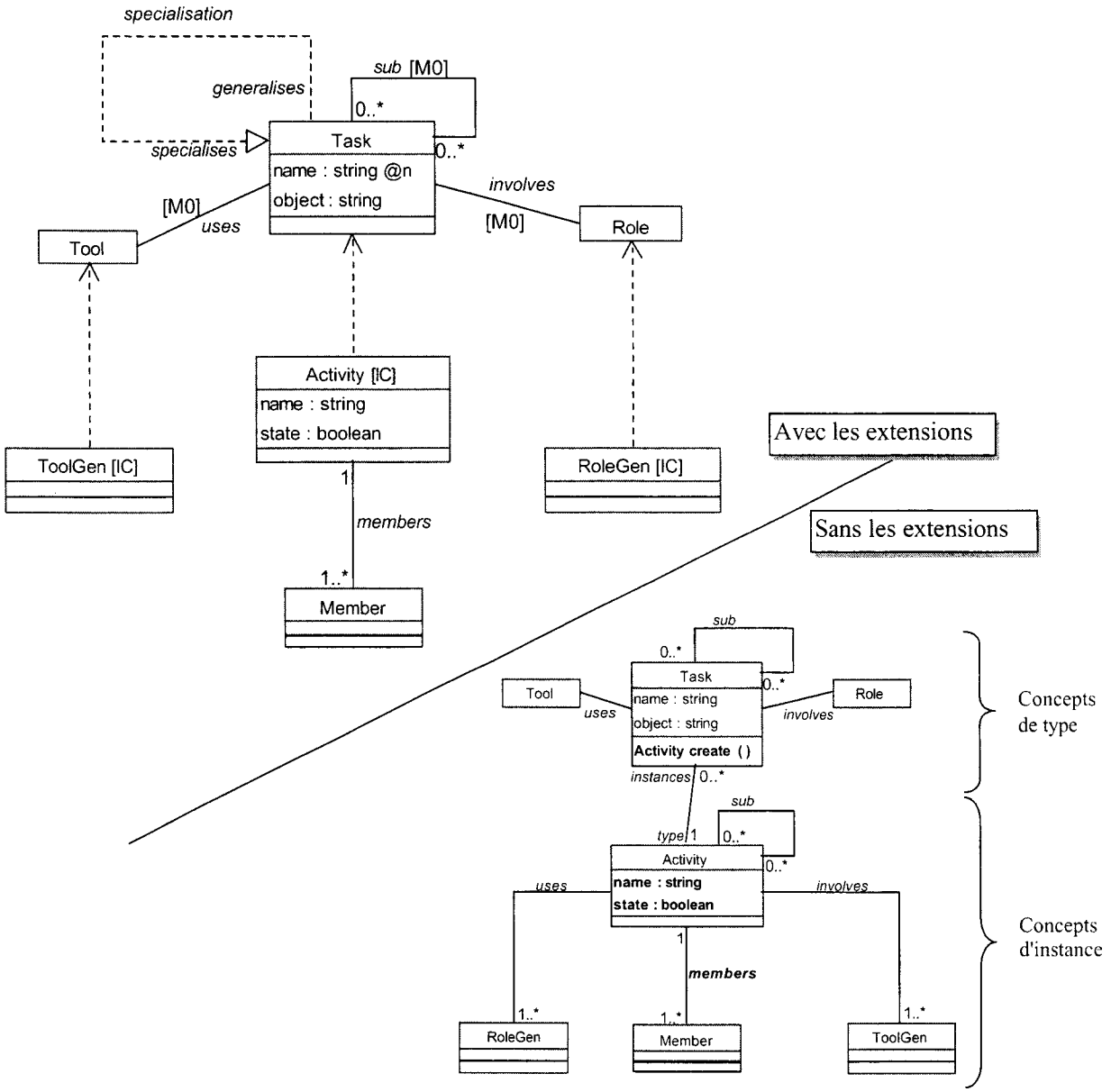


figure 70. Définition des concepts Task et Activity avec les extensions graphiques

4.3.3 Modification sur le support d'instanciation Corba*

Chaque extension modifie les règles de projection vers le support d'instanciation Corba. Il y a deux niveaux de modifications : les interfaces IDL et l'implémentation/traitement associé.

4.3.3.1 Le concept d'instance

Nous reprenons le style d'expression des règles de projection présentes dans les spécifications du MOF. Dans l'annexe A figurent les définitions précises des règles.

De manière analogue à la *class* MOF normale, la *class_IC* va générer deux interfaces IDL : une fabrique et une définition des points d'entrée des instances.

Class_IC Template

// forward definition
interface <*Class_IC_Name*>**Class**;
interface <*Class_IC_Name*>;

<< CLASS_IC TEMPLATE>>
<< M0 INSTANCE TEMPLATE>>

La fabrique reprend la même structure associée à *Class* à la différence qu'apparaît dans l'opération de création, un paramètre supplémentaire pour chaque concept de type associé à la *class_IC*. Suivent ensuite les paramètres pour chaque attribut non dérivé (règle définie par *Class*). Il faut rajouter dans cette liste de paramètre, les attributs non dérivés issus des CTs notés [M0].

A travers notre expérience du MOF, nous avons conclu que l'affectation d'une valeur à chaque paramètre/attribut (dans l'opération de création de la fabrique) est contraignante. Pour cela, nous avons défini une opération de création qui prend simplement le nom en paramètre à la place de tous les paramètres/attributs. Le nom passé en argument sera affecté à l'attribut noté @n. Il est, pour cette raison, important d'annoter l'attribut de nommage le plus souvent possible. Nous ajoutons cette opération de création réduite aux Class et aux Class_IC. Le langage IDL ne supportant pas la surcharge, le nom de cette opération est suffixé par *_name*.

Class_IC Template

interface <*Class_IC_Name*>**Class :**
// if Class_IC has no super-Classes_IC
Reflective::RefObject
// else for each super-Classes_IC
<*SuperClass_IC*>**Class,** ...
{
<*Class_IC_Name*> **create_**<*class_IC_name*> (
// for each Class realised by this Class_IC
in <*ClassName*> <*ClassName*>*_type*, ...
// for each non-derived direct or inherited attribute
in <*AttributeType*>[<*CollectionKind*>] <*attributeName*>, ...
// for each non-derived direct or inherited attribute of associated classes
// which is designed as present in the Class_IC
in <*Class_IC_AssociatedToTheAttributeType*> [<*CollectionKind*>]
<*attributeName*>, ...
) **raises (Reflective::MofError);**

// if an attribute is @n annotated
<*Class_IC_Name*> **create_**<*class_IC_name*>**_name (**

```
        // for each Class realised by this Class_IC
        in <ClassName> <ClassName>_type, ...
        in <@n_AttributeType> name
    ) raises (Reflective::MofError);

};
```

Class Create Template - Extended

```
// if an attribute is @n annotated
<ClassName> create_<class_name>_name (
        in <@n_AttributeType> name  ) raises (Reflective::MofError);
```

L'interface IDL d'une instance de classe, quand celle-ci est un concept de type, se voit ajouter deux opérations de création/instanciation qui correspondent aux opérations complète et réduite de la fabrique du CI associé. Pour chaque opération, on retire le paramètre où la classe est le type de celui-ci.

Instance (of a Class) Template - Extended

```
// if a class_IC is associated with the Class
// and if the class is designated as a constructor
<Class_IC_Name> newInstance_<class_IC_name> (
        // for each Class, different of this Class, realised by this Class_IC
        in <ClassName> <ClassName>_type, ...
        // for each non-derived direct or inherited attribute
        in <AttributeType>[<CollectionKind>] <attributeName>, ...
        // for each non-derived direct or inherited attribute of associated classes
        // which is designated as present in the Class_IC
        in <Class_IC_AssociatedToTheAttributeType> [<CollectionKind>]
                                                <attributeName>, ...
    ) raises (Reflective::MofError);

// if a class_IC is associated with the Class
// and if the class is designated as a constructor
// and if an attribute is @n annotated
<Class_IC_Name> newInstance_<class_IC_name>_name (
        // for each Class, different of this Class, realised by this Class_IC
        in <ClassName> <ClassName>_type, ...
        in <@n_AttributeType> name
    ) raises (Reflective::MofError);
```

4.3.3.2 Les éléments M0

L'interface IDL des instances de CI se différencie de l'interface IDL des instances de CT d'abord par la présence des types de définitions. Une opération de consultation par type est créée. Elle est suffixée par *_type*. Nous n'avons pas permis le changement de type de définition après la création de l'élément M0 (une opération *set*)[65].

M0 Instance Template

```
interface <Class_IC_Name> :
  <Class_IC_Name>Class,
// for each super-Classes_IC
<SuperClass_IC>, ...
{
        // for each type Concept associated,
        <ClassName>  <ClassName>_type( ) raises (Reflective::MofError);

        << GENERIC OPERATIONS TEMPLATE>>

        // for each Attribute, Reference, Operation contained in this Class_IC
        // generate the appropriate IDL
        << ATTRIBUTE OLD TEMPLATE (V1.3)>>
        << REFERENCE OLD TEMPLATE (V1.3)>>
        << OPERATION TEMPLATE>>

        // for each [M0] Attribute, [M0] Reference contained in the type concepts
        // generate the appropriate IDL
        << ATTRIBUTE IC TEMPLATE>>
        << REFERENCE IC TEMPLATE>>
 };
```

Les attributs et les références d'une *Class_IC* apparaissent dans l'interface IDL de la même manière qu'ils le font pour une *Class*. Par contre, ils ne disposent pas des extensions pour la multiplicité. Les opérations, quant à elles, n'ont aucune différence de projection.

Les templates *ATTRIBUTE IC* et *REFERENCE IC* correspondent aux attributs et références de CT notés [M0]. Ces templates sont les règles de répercussion de ces caractéristiques sur les éléments M0. L'instanciation du type de valeur (obtention du type primaire) est la première transformation par rapport aux templates *ATTRIBUTE/REFERENCE* habituels. La définition de points d'accès secondaires est la deuxième transformation. Les opérations qui résultent de cette transformation, suffixées *_byType* ou *_byName*, permettent de filtrer la liste des valeurs par le type secondaire ou par le nom de celui-ci. Par exemple l'invocation de `involves_byName ("etudiant")` sur une activité de type *TPA* renvoie la liste des rôles de type *Etudiant*. Une alternative est proposée avec le template *GENERIC OPERATIONS*. Il offre un ensemble d'opérations génériques qui s'appliquent non

[65] Les mécanismes d'intercession évoqués précédemment nous semble déjà suffisant. Proposer le changement de type de définition correspond à une souplesse peut-être trop grande. Néanmoins cela peut constituer une extension future.

plus à une caractéristique [M0!] mais à toutes les caractéristiques [M0]. Ces opérations génériques ne sont destinées qu'aux caractéristiques héritées des concepts de type (et pas exclusivement aux points d'entrée secondaires). Par exemple, l'invocation `get("involves","etudiant")` sur l'activité précédente est équivalent à `involves_byName ("etudiant")`.

Generic Operations Template

```
// this is a set of generic operations for secondary entry points
// different from Reflective package
// these operations deal with [M0] structuralFeatures.
// valueType cannot be a datatype
// this set is present if there is at least one [M0] StructuralFeature

RefObject get (in string featureName, in string secondaryTypeName )
    raises (Reflective::NotSet, Reflective::MofError);

void set (in string featureName, in string secondaryTypeName,
    in RefObject new_value ) raises (Reflective::MofError);

void unset (in string featureName, in string secondaryTypeName,
    in RefObject new_value ) raises (Reflective::MofError);

void add (in string featureName, in string secondaryTypeName,
    in RefObject new_element ) raises (Reflective::MofError);

void modify (in string featureName, in string secondaryTypeName,
    in RefObject old_element, in RefObject new_element )
      raises (Reflective::NotFound, Reflective::MofError);

void remove (in string featureName, in string secondaryTypeName,
    RefObject old_element ) raises (Reflective::NotFound, Reflective::MofError);
```

Attribute IC Template

```
// if Attribute visibility is private or protected no IDL
// is generated

// GET
// if lower = 0 and upper = 1
<Class_IC_AssociatedToAttributeType> <attribute_name> ( )
      raises (Reflective::NotSet, Reflective::MofError);
```

```
// if lower = 1 and upper = 1
<Class_IC_AssociatedToAttributeType> <attribute_name> ( )
    raises (Reflective::MofError);

// if upper > 1
<Class_IC_AssociatedToAttributeType><CollectionKind> <attribute_name> ( )
    raises (Reflective::MofError);

// if attribute is [M0!]
<Class_IC_AssociatedToAttributeType> <attribute_name>_byType
    ( in <AttributeType> type ) raises (Reflective::NotSet, Reflective::MofError);
<Class_IC_AssociatedToAttributeType> <attribute_name>_byName
    ( in string name ) raises (Reflective::NotSet, Reflective::MofError);

// SET
// if upper = 1 and is_changeable
void set_<attribute_name> (in <Class_IC_AssociatedToAttributeType> new_value)
    raises (Reflective::MofError);

// if upper > 1 and is_changeable
void set_<attribute_name> (
    in <Class_IC_AssociatedToAttributeType><CollectionKind> new_value )
    raises (Reflective::MofError);

// if attribute is [M0!]
void set_<attribute_name>_byType ( in <AttributeType> type,
    in <Class_IC_AssociatedToAttributeType> new_value )
    raises (Reflective::MofError);
void set_<attribute_name>_byName ( in string name,
    in <Class_IC_AssociatedToAttributeType> new_value )
    raises (Reflective::MofError);

// UNSET
// if lower = 0 and upper = 1 and is_changeable
void unset_<attribute_name> ( )
    raises (Reflective::MofError);

// if attribute is [M0!]
void unset_<attribute_name>_byName ( in string name )
    raises (Reflective::MofError);

// ADD
// if upper > 1 and is_changeable
void add_<attribute_name> (
```

```
        in <Class_IC_AssociatedToAttributeType> new_element)
        raises (Reflective::MofError);

    // if attribute is [M0!]
    void add_<attribute_name>_byType ( in <AttributeType> type,
        in <Class_IC_AssociatedToAttributeType> new_element)
    void add_<attribute_name>_byName ( in string name,
        in <Class_IC_AssociatedToAttributeType> new_element)

    // ADD BEFORE
    ...
    // ADD AT
    ...
    // MODIFY
    ...
    // REMOVE
    ...
```

4.3.3.3 La gestion des caractéristiques des éléments M1

Chaque entrée d'une caractéristique [M0] d'un élément M1, correspond à une sous-caractéristique ou même à un nouveau point d'entrée ([M0!]). Pour cette raison, l'entrée doit fournir au minimum un nom, un type et une multiplicité (cf 4.3.1.4). Précédemment nous avons indiqué que si aucune caractéristique de l'entrée n'est désignée comme la multiplicité (à l'inverse du nom et du type), il faut lui en associer une. C'est l'objet des extensions aux templates ATTRIBUTE et REFERENCE destinés aux instances de classes.

```
Attribute/Reference Template - Extended

// if Attribute or Reference is [M0] annotated
// if AttributeType or ReferenceType does not provide a multiplicity information

// GET
<SelectedDataType> <attribute_name>_multiplicity ( )
      raises (Reflective::NotSet, Reflective::MofError);
<SelectedDataType> <attribute_name>_multiplicityAt
        ( in unsigned long position)
        raises (Reflective::NotSet, Reflective::MofError);

// SET
void set_<attribute_name>_multiplicity ( <SelectedDataType> multiplicity )
      raises (Reflective::NotSet, Reflective::MofError);
void set_<attribute_name>_multiplicityAt
        ( in unsigned long position, <SelectedDataType> multiplicity)
```

```
        raises (Reflective::NotSet, Reflective::MofError);
void set_<attribute_name>_multiplicityFor
        (in < AttributeType> element, <SelectedDataType> multiplicity)
        raises (Reflective::NotSet, Reflective::MofError);
```

4.3.3.4 L'implémentation générée

L'implémentation associée à la projection IDL reprend chacun des points évoqués dans la partie 4.2.2.

4.4 CAST-MOF

Nous avons conçu CAST pour qu'il fournisse des services d'adaptation pour le MOF au niveau des éléments M1. Nous l'avons aussi conçu pour qu'il fonctionne avec les précédentes extensions du MOF, c'est-à-dire une adaptation au niveau M0. Ces extensions ont été présentés comme utiles dans le cadre de la conception de système (c'est le cas d'une nouvelle version de DARE). Mais ce n'est pas leur seul intérêt : c'est sur elles que va se reposer M-CAST pour adapter des éléments M-zéro. Ainsi que nous l'avions expliqué, si une norme sur le niveau M0 vient à être créée nous modifierons CAST pour l'y adapter. Pour l'instant une telle norme n'existe pas. Il nous a semblé qu'étendre le MOF était une solution judicieuse car le fait que le MOF soit le seul vrai standard pour le niveau M2 et M1 en fait le meilleur candidat pour une standardisation du niveau M0 : le niveau M0 est forcément lié aux niveaux M2 et M1.

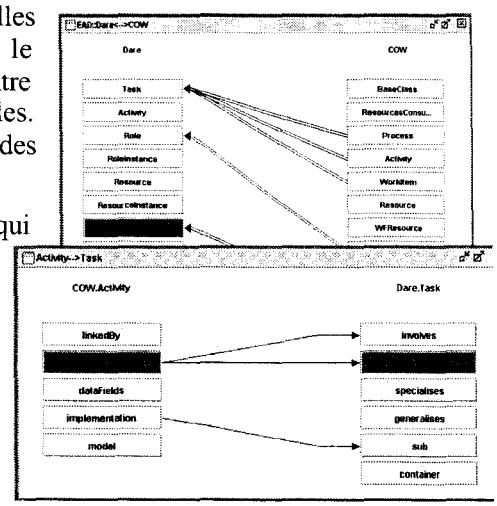
5 Le framework CAST::MOF-Corba

Nous allons maintenant appliquer le pattern CAST, structure générique pour les services d'adaptation, sur le MOF et son support Corba (M1 et M0). La figure 32 du chapitre 3 montre que, conceptuellement, le service d'adaptation est composé d'une partie spécifique aux méta-modèles des systèmes coopérants et une partie fixe, le paquetage CAST, qui constitue le gestionnaire d'adaptateurs.

Note : Dans le reste de cette partie, nous considérons le langage Java comme langage d'implémentation des classes UML et/ou des interfaces IDL.

5.1 Le déploiement du service d'adaptation

Une question importante est de savoir où se trouve le service d'adaptation. En fait peu importe son emplacement, car le support MOF-Corba est distribué et donc accessible de n'importe où. Néanmoins, nous pensons qu'à terme la génération de telles passerelles sera le travail des nouveaux métiers évoqués dans le chapitre 2 qui ont pour objectif de faciliter la coopération entre systèmes informatiques dans le cadre des organisations virtuelles. Les services d'adaptation pourraient donc se trouver sur des serveurs dédiés.



Le service d'adaptation est généré à partir d'un outil qui reprend les concepts de CAST. Il faut indiquer les liens d'adaptation. Pour cela, l'outil doit proposer un éditeur de liens reprenant le formalisme graphique présenté dans le chapitre 4. Dans le chapitre 6 nous verrons un exemple d'éditeur. L'outil, alimenté par ces liens, génère les paquetages *X_Adapter* et *X_Filter*.

5.2 Le paquetage CAST

L'implémentation des différentes classes UML du paquetage CAST ne pose pas de réel problème. La seule difficulté provient de la nécessité ou non de l'interface IDL pour les éléments du paquetage. En d'autres termes, on se pose la question pour chaque classe UML du modèle conceptuel CAST, si ses instances ont besoin d'être accessible sur le réseau, c'est-à-dire d'être présentes sur le bus Corba ? Ou l'implémentation en Java (sans interface IDL associée) est-elle suffisante ?

Le service. Le service d'adaptation (une instance de la classe *AdaptationService*) n'a pas besoin d'avoir un interfaçage Corba. En fait, seuls les adaptateurs, les filtres ou les référentiels d'importation (cf. plus loin) se connectent à lui. Ce sont à chaque fois des objets créés par le service lui-même. Ces accès, qui s'effectuent localement, ne nécessitent donc pas de créer une interface IDL pour le service d'adaptation. Néanmoins, on peut prévoir d'administrer le service d'un poste distant dans le cas où celui-ci aurait été mal manipulé et contiendrait des erreurs (sémantiques) impossibles à réparer à partir des systèmes coopérants. Le raisonnement s'applique de manière similaire pour les tables d'accès, des adaptateurs et des traitements. Les accès ne nécessitent pas un interfaçage Corba, mais il est néanmoins intéressant d'envisager une possibilité d'administration. La création d'une interface IDL pour chacun de ces éléments est pourtant freinée par le fait que l'administration implique des restrictions d'accès aux opérations du service d'adaptation et de ses tables. Il faut donc mettre en place une protection. Nous n'avons pas, pour l'instant, étudier suffisamment les techniques existantes sur ce type de problème et donc nous n'avons pas conçu ni implémenter cette protection. La proposition que nous développerons dans nos prochains travaux serait de définir une interface IDL `Monitor` qui reprendrait l'ensemble des opérations du service d'adaptation et des tables. `Monitor` serait le type IDL d'objets Corba fournis par une opération `get_aMonitor` d'un objet serveur (`AdaptationService`) qui demanderait en paramètres un login et un mot de passe. Les opérations d'un objet `Monitor` auraient un effet ou non selon les droits du profil identifié par le login.

Les adaptateurs. La classe *Adapter* requiert une interface IDL car les adaptateurs ont besoin de se faire reconnaître comme tels sur le support Corba. *Adapter* définit une opération pour connaître la source. On peut imaginer (cf. figure 71) que la société LECS (COW) contient un *process*, adaptation d'une tâche issue d'une autre société (*société X*) que HLC utilisant pourtant le système DARE. Si le système d'HLC importe cet élément, le service d'adaptation entre HLC et LECS, plutôt que d'adapter le précédent *process* en tâche, peut directement fournir la tâche à HLC puisqu'il s'agit d'un objet gérable par son système (DARE).

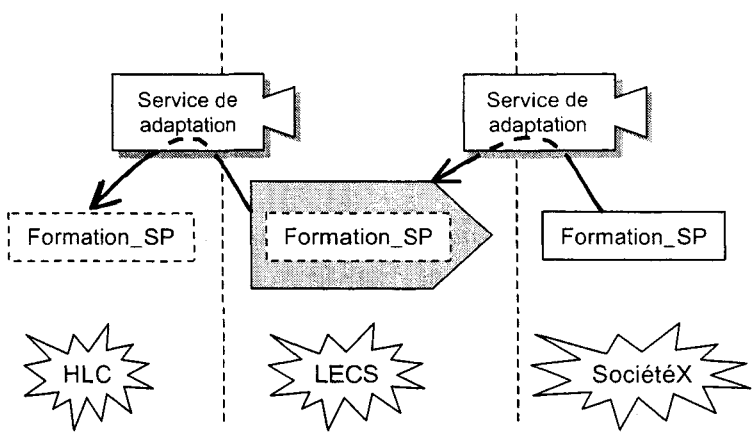


figure 71. Un AS doit reconnaître les adaptateurs

Les filtres. Un objet filtre ne peut pas être utilisé par un objet traitement n'appartenant pas au service d'adaptation dont le filtre provient. La classe *Filter* ne requiert pas d'interface IDL.

Les traitements. Enfin le paquetage *processingObjectRepository* contient des classes dont les instances sont utilisées seulement par les adaptateurs. Les objets traitements n'existent que pour

apporter une certaine flexibilité au service d'adaptation et n'ont pas à être visible de l'extérieur. Le paquetage *processingObjectRepository* et les classes qu'il contient n'impliquent pas la création d'interface IDL.

5.3 Les paquetages X_Adapter *

L'étude de la projection des paquetages *X_Adapter* vers le support MOF-Corba se compose en deux parties : la classe *Repository*, c'est-à-dire le référentiel, et l'ensemble des classes correspondantes aux entités du méta-modèles.

5.3.1 Le référentiel

Chaque système dispose d'un référentiel de modèles. Nous avions indiqué dans le chapitre 4, que pour notre solution CAST, une contrainte sur le système était qu'il puisse gérer plusieurs référentiels avec une fonction d'importation et une d'exportation. Les modèles externes qu'un système doit importer sont intégrés grâce à un référentiel supplémentaire. Ce référentiel ne contient que des modèles d'UN autre système. Nous le désignerons par "référentiel d'importation" (RIm).

Dans les chapitres 3 et 4, le référentiel est un raccourci de "interface du référentiel". Il est défini comme contenant des opérations pour connaître toutes les instances d'une entité, pour instancier une entité, ... Il est aussi indiqué que cette définition n'a qu'un but pédagogique, car le référentiel fait partie de l'opérationnalisation et il peut donc être d'une multitude de formes. Par conséquent, la structure et surtout le comportement du RIm sont donc particuliers à chaque support.

5.3.1.1 Le référentiel en MOF/Corba

Ce que nous désignons par interface de référentiel pour le MOF et son support Corba sont les objets Corba représentant le package, les fabriques d'instances d'entité et les fabriques de liens (cf. figure 72).

L'ensemble de ces objets permet de créer des instances et permet de connaître les instances existantes.

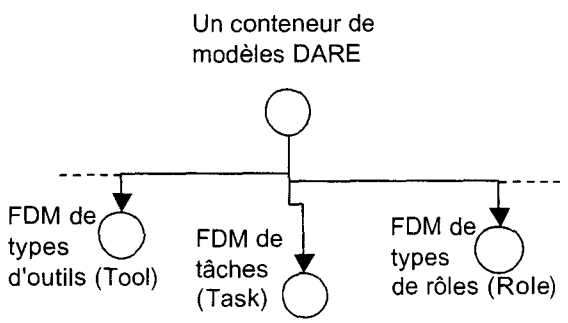


figure 72. Le référentiel de HLC (DARE)

5.3.1.2 Le référentiel d'importation

Le référentiel d'importation est composé des mêmes types d'objets. Par contre leur comportement est différent :

- Les fabriques associées aux concepts de type ne peuvent créer d'instances.
- Les fabriques de liens veillent à reporter les liens dans le référentiel importé.

Fabriques de concepts de type. La création d'instances de concepts de type, c'est-à-dire la création d'éléments M1, pour le RIm est inappropriée. Créer un élément M1 dans le RIm correspondrait sur notre exemple HLC-LECS à la situation suivante : un employé d'HLC crée un type de processus (un type réel et pas une adaptation de tâche) dans le référentiel de LECS. Il ne s'agit plus ici de lier des éléments de modèles de systèmes différents mais d'ajouter directement des éléments dans un référentiel de modèles d'un système à partir d'un autre système (comme le ferait un spécialiste du premier système). Ceci sort de notre contexte de coopération. Même si cette opération semble faisable et séduisante, elle implique un autre type de relation entre les entreprises qu'il faudrait étudier précisément et suppose probablement des contraintes d'autorité qu'il serait nécessaire de prendre en compte. Dans notre contexte, si LECS doit intégrer un nouveau modèle (réel et non adapté) de processus dont la motivation provient d'une coopération, ce sont ses employés qui créent ce modèle à la suite de directives ou souhaits émis par le partenaire.

Fabriques de liens. Les liens créés entre deux éléments M1 issues de systèmes différents doivent apparaître dans chacun des systèmes. Ceci est du à notre contrainte de visibilité. Ainsi un lien inter-référentiel existe dans deux fabriques. L'existence, en MOF, de ces gestionnaires de liens va entraîner une autre contrainte sur les systèmes qui vient compléter la première : les systèmes doivent pouvoir gérer plusieurs référentiels ET gérer des liens inter-référentiels. Un lien entre deux éléments M1 apparaît dans chacun des référentiels concernées (un seul si les instances sont du même référentiel).

5.3.2 Les adaptateurs

Dans le paquetage *X_Adapter* se trouvent, en plus du RIm, les classes adaptatrices. Celles-ci héritent des classes correspondantes aux CTs ou Cis qu'elles adaptent. L'implémentation de cette spécialisation consiste en une redirection vers les objets traitements. L'héritage n'a pas pour objectif une extension de l'interface mais une simple redéfinition de l'implémentation des opérations existantes. Dans le cadre du MOF/Corba, il n'y a donc pas de nouvelles interfaces IDL. Néanmoins, les classes d'adaptation héritent aussi de la classe *Adapter*. Il est important qu'un adaptateur soit reconnu comme tel sur le support Corba. Pour cette raison, la classe *Adapter* a entraîné la création de l'interface IDL `Adapter` et implique, pour chaque entité du méta-modèle, la création d'une interface IDL héritant de l'interface IDL "instance" du concept de type ou d'instance correspondant et de l'interface IDL `Adapter`. Suit la définition de l'interface IDL pour l'entité *Task*.

```
module DARE_Adapter {
...
interface Task :
  DARE.Task, CAST.Adapter
  {
  };
... };
```

La classe Java associée à ce type d'interface consiste à implémenter l'accès à la source et la redirection pour chaque "opération" définie du concept.

5.3.3 Les filtres

Le cas des filtres ne suit pas le même schéma que les adaptateurs. Les paquetages *X_Filter* ont pour objectif de limiter l'accès à certaines caractéristiques. Ce type de paquetage contient des classes (et pas de référentiel) spécialisant les classes/entités sans les étendre. Il n'y a donc pas de génération de nouvelles interfaces IDL mais par contre, un paquetage Java est créé. Celui-ci définit des classes Java qui implémentent les interfaces IDL des entités avec des objectifs de filtrage. L'implémentation permet ou non de consulter ou d'affecter la valeur d'une caractéristique selon les droits d'accès de l'élément.

5.3.4 Les traitements

Chaque service dispose d'un paquetage où ils stockent ses classes "traitements". Chaque classe correspond au traitement d'adaptation d'un concept. La classe n'hérite pas de celle correspondante à l'entité, mais redéfinit chacune de ses opérations avec un paramètre supplémentaire : la source (un objet filtré). Le code des opérations est soit la reprise directe de œ qui a été défini dans les liens d'adaptations (si c'était en langage Java) ou une traduction (si ce n'était pas défini en Java).

6 Conclusion

CAST définit un modèle conceptuel de service d'adaptation. Pour l'implémenter, en vue d'une validation, nous avons choisi le support de représentation de modèles MOF. Il est le seul "standard" de ce type actuellement. Son objectif est de normaliser l'accès informatique des méta-modèles. Il reprend les concepts d'UML et fournit un accès "normalisé" et distribué via Corba, particulièrement intéressant

dans notre contexte de coopération de système. Le support MOF-Corba constitue la cible de notre projection de CAST (M-CAST). Pour économiser du temps dans le développement d'un système flexible (structuré sur trois niveaux), nous avons défini des extensions au MOF pour générer la couche M-zéro. En l'absence de norme pour le niveau M-zéro, M-CAST se base sur ces extensions pour adapter les instances de modèles. Ce chapitre se conclut sur les définitions d'interface IDL concernant les services d'adaptation et les implications d'implémentation.

Le chapitre suivant, 6, concerne une implémentation réelle de M-CAST, et aborde pour cela des aspects relatifs à l'utilisation de M-CAST. Le prototypage est l'un des aspects les plus importants : une liaison conceptuelle ou structurelle erronée ou incohérente peut avoir de lourdes conséquences sur le partenariat des entreprises correspondantes. Le besoin d'une phase de prototypage nous a amené à développer un outil MOF dynamique.

Chapitre 6

RAM3
Rapid Manipulation of MOF Metadata : Un environnement de développement MOF

Une erreur dans la conception d'un service d'adaptation (c'est-à-dire dans la définition des liens d'adaptation) peut être lourde de conséquences d'un point de vue commercial. Dans le cas du scénario HLC-LECS et de l'exemple de la modification non souhaitée de types de processus (chapitre 4, section 2.3.3) l'erreur peut entraîner des activités vides ou incohérentes et causer le mécontentement des clients. L'outil M-CAST ne peut donc pas être véritablement utilisable et intéressant sans avoir de possibilités de prototypage. Les services d'adaptation étant essentiellement constitué "d'objets MOF", le prototypage de ces services nécessite un véritable environnement de développement MOF avec la possibilité de "manipuler" simultanément des modèles issus de méta-modèles différents. Prototypage rime généralement avec modification dynamique : ici la possibilité de modifier dynamiquement le service d'adaptation et d'en voir les répercussions. En d'autres termes, l'environnement doit disposer de mécanisme d'intercession entre les modèles et leurs méta-modèles. Actuellement aucun outil MOF ne dispose de telles caractéristiques (et ne dispose pas non plus d'une liaison avec le niveau M-zéro). C'est une des deux raisons pour laquelle nous avons développé l'outil RAM3. Il se révèle être au final un environnement de développement de référentiels MOF avec de puissantes capacités de prototypage et intégrant le niveau M-zéro. La deuxième raison de son développement est que les précédentes caractéristiques sont aussi très profitables pour le prototypage de référentiel de modèles dans le cadre du développement d'un système à trois niveaux : RAM3 est aussi un environnement de méta-systèmes compatibles MOF.

Ce chapitre commence par le détail du cahier de nos exigences vis-à-vis d'un outil MOF support à M-CAST. Ce cahier montre qu'aucun outil MOF n'offre un réel support pour le prototypage de services d'adaptation et même de référentiel de modèles. La réponse à ces problèmes par notre outil RAM3 constitue la partie suivante de ce chapitre. Nous exposons ensuite de manière précise l'architecture de RAM3. Celui-ci ayant demandé un grand travail de développement, nous terminons sur l'état d'avancement de M-CAST.

Note : RAM3 est un outil de méta-modélisation MOF. Il peut donc éditer tout type de méta-modèles, modèles (et éléments M-zéro). Les exemples cités tout au long de ce chapitre, dont le but est de mieux illustrer nos propos, mettent en jeu soit des éléments de niveau M2 (concepts/entités) soit des éléments de niveau M1 (modèles) ou encore des éléments M-zéro. Tout les exemples sont issus du scénario HLC-LECS : entités de DARE ou de COW pour les exemples d'entités, modèles de HLC ou LECS pour les exemples de modèles et leurs instances pour les exemples d'éléments M-zéro. Parfois, des annotations comme ex. M1 *seront présentes. La précédente indique qu'il s'agit d'un exemple de modèle. Cette note et ces annotations ont pour but d'indiquer clairement que RAM3 et (M-)CAST ne sont attachés à aucun méta-modèle (comme DARE ou COW) ni modèle (comme HLC ou LECS) en particulier.*

1 Besoins pour M-CAST

L'application de CAST sur le support MOF/Corba suppose une maîtrise logicielle complète du MOF. Le développement d'un service d'adaptation en lui-même, c'est-à-dire quelque soit l'environnement d'application utilisé, peut s'effectuer à travers une approche itérative (incrémentale) pour faciliter la définition des liaisons d'adaptation. L'utilisation définitive du service d'adaptation dans un contexte industriel nécessite une phase de test. La maîtrise logicielle du MOF, l'approche itérative et les phases de tests laissent supposer que la conception de M-CAST est conséquente. Pour cette raison, il est préférable que M-CAST se repose sur un outil MOF, c'est-à-dire que M-CAST utilise ou s'intègre à celui-ci. Les outils MOF offrant des fonctionnalités différentes, les trois précédents aspects (maîtrise logicielle du MOF, ...) établissent une liste des caractéristiques nécessaires de l'outil MOF.

1.1 Développement de M-CAST

L'architecture d'M-CAST peut grossièrement se diviser en quatre fonctions principales ayant chacune un objectif précis (dans l'ordre chronologique d'utilisation) :

- Chargement des méta-modèles "coopérants".
- Éditeur de liaisons entre les entités et leurs caractéristiques.
- Génération des interfaces IDL pour les paquetages *X_Adapter*.
- Génération/Association du code d'implémentation (Java).

Le chargement et l'éditeur implique la connaissance d'un format de persistance des méta-modèles MOF (XMI, MODL, ou autres), l'implémentation d'une procédure de lecture de ce format et la création de structures mémoire (ex : arborescences d'objets) correspondantes aux méta-modèles. Ces fonctionnalités ne sont pas simples à développer vu le nombre conséquent de propriétés pour une classe, un attribut ou une référence MOF. L'utilisation d'une API déjà existante, qui charge "des" méta-modèles en mémoire, semble raisonnable. Cette API constitue la première caractéristique nécessaire de l'outil MOF.

1.2 Définition "progressive" de liens d'adaptation

La définition de liens d'adaptation n'est pas particulièrement intuitive. Sa pratique ne s'apparente que peu à la transformation de modèles. Donc, même pour un adepte de la transformation de modèles, le premier contact peut dérouter. Le développeur peut être aussi rendu perplexe pour des méta-modèles complexes ou forts dissemblables. Pour faire face à cette complexité, une approche "itérative", c'est-à-dire une définition de liens entrecoupée d'une multitude de (petits) tests est appréciable. Sur notre exemple DARE-COW, la simple liaison entre *Task* ← *Process* peut être immédiatement suivie d'un test : le développeur peut voir apparaître les types de processus de COW sous forme de tâches dans DARE. Cette exemplification apporte plus de repères au développeur et rend plus intuitive la définition des liaisons suivantes. Pour que les tests successifs ne rendent pas cette méthode trop longue et fastidieuse, il faut pouvoir ajouter/modifier/supprimer les liens sans devoir

ex. M2

arrêter le service d'adaptation, le recompiler, le relancer et le reconnecter aux systèmes. Il est plutôt intéressant que les exemples issus des tests précédents se voient "raffinés" grâce aux nouveaux liens. Par exemple, la liaison *involves* ← *performers* entraîne l'apparition de valeurs pour les rôles des *Task(Process)* apparues lors du test précédent. Les modifications des liaisons doivent être dynamiques. Le paquetage des objets traitement de CAST permet cette dynamicité. Néanmoins l'objectif que nous avions fixé (cf. Chapitre 3) pour ce paquetage apparaît quand le service d'adaptation est en réelle utilisation, c'est-à-dire lorsque les liaisons sont déjà toutes définies. Les modifications effectuées sont de l'ordre de la finition. La situation est différente pour le cas de l'approche itérative. Sur le premier exemple, les objets adaptés n'ont pratiquement pas de valeurs pour chacune de leur caractéristique. Ceci peut entraîner des erreurs lors de la consultation des modèles à l'aide de l'éditeur[66]. Dans l'approche itérative, il ne faut non plus être trop contraint. Si des erreurs dans un modèle risquent facilement de faire planter l'éditeur de modèles ou de déclencher des exceptions (avec affichage d'avertissement) dans celui-ci qui interdirait l'accès aux parties erronés (et donc supprimerait la possibilité d'effectuer une correction) le développeur sent ses libertés diminuées. Une solution séduisante est d'avoir un outil MOF qui permet de charger plusieurs méta-modèles simultanément et d'éditer de manière brute des modèles de ces derniers, c'est-à-dire sans contrainte sur les valeurs. Les menaces d'échec d'exécution de l'éditeur ou d'accès bloqués sont ainsi supprimées. De plus, cette solution évite de travailler directement avec les systèmes coopérants donc 1) de naviguer sans cesse de l'un à l'autre, 2) de risquer de venir altérer le contenu existant des systèmes.

1.3 Débogage d'un service d'adaptation

Même si la définition des liens d'adaptation n'a pas suivi une approche itérative, la mise en place du service d'adaptation, correspondant à une utilisation réelle, nécessite au préalable une phase de tests : des erreurs ou incohérences au sein des liaisons d'adaptation peuvent être dramatiques pour les deux sociétés coopérantes. Il faut donc vérifier qu'il n'y a pas, dans les liens d'adaptation, des incohérences ou des sources de blocage.

Par exemple, le lien *Task.sub*←*Activity* peut être source d'incohérences. Lorsqu'on ajoute une sous-tâche à une *Task(Activity)* celle-ci peut être stockée dans la liste des sous-activités si l'activité adaptée est composite (i.e. référence, à travers *model*, un type de processus) ou sinon dans la liste des *workitems*. Si cette subtilité est mal exprimée, des incohérences peuvent apparaître. ex. M2

Une tâche ajoutée *ComputerVocab* à l'activité *Spanish* peut par exemple apparaître dans la liste des *workitems*. Comme *Spanish* référence, à travers *model*, *SpanishDef* (un type de processus), l'éditeur n'affiche pas ses *workitems*. L'ajout du *Process(Task) ComputerVocab* n'a aucun effet pour le contexte de COW. Pire, à l'exécution *ComputerVocab* ne sera pas instancié. ex. M1

Si l'incohérence n'a pas été notée pendant la modélisation, il sera encore plus difficile de la détecter à l'exécution car :

1) Avant l'exécution du modèle, un certain nombre de liens (entre éléments M1) ont été définis constituant autant de sources hypothétiques "de l'erreur" pour le développeur.

2) Les instances des modèles ne sont pas affichées aussi explicitement que les éléments de modèles : ils sont une source de coordination mais ne sont pas affichés telles quelles. Le débogage se fait en "aveugle".

Des erreurs plus directes sont possibles. Les ensembles restreints de valeurs en sont de bons candidats : les valeurs autorisées d'une caractéristique sont un sous-ensemble des valeurs permises par le type de la caractéristique (ex : un entier de 0 à 10, une chaîne de caractères avec des caractères interdits, ...). Les méthodes d'affection (*set*) de ce type de caractéristique sont les gardiens du respect de ces contraintes. Par exemple, pour des raisons historiques, le nom d'une tâche (DARE) ne peut

[66] À la création d'une tâche, DARE crée des valeurs de départ pour certaines caractéristiques. L'éditeur de DARE suppose donc une valeur pour ces caractéristiques.

contenir d'espace[67]. Ce n'est pas le cas de COW. Si un lien *Task.name←Activity.name* est fait sans filtrage, le nom d'une *Task(Activity)* peut être erroné http://www.freetelecom.com (la méthode `set_name` de `Activity` ne gère pas l'interdiction d'espace). Si sur ce cas, il n'y a pas de blocage possible, cela peut être différent avec d'autres situations : par exemple, un potentiomètre graphique est attaché à une caractéristique pour laquelle la valeur (entière) ne peut dépasser 10. Si la valeur pour un élément adapté dépasse 10, la coordonnée du potentiomètre peut être trop grande. Cette saturation peut générer une erreur ou un blocage non prévue et empêcher l'affichage complet de l'élément. Il est difficile dans ce cas de voir que l'erreur provient de cette caractéristique. Des valeurs erronées peuvent aussi être sources de problème à l'exécution des instances de modèles.

ex. M2

L'utilisation d'un éditeur générique de modèles MOF sera donc aussi appréciable pour la recherche d'erreurs dans la phase de test finale.

Que ce soit pour une définition progressive de liens d'adaptation ou pour des phases de test, il est intéressant d'intégrer la notion d'adaptateur dans l'éditeur. Ainsi quand on ajoute une sous-tâche à une *Task(Process)*, cette dernière est facilement reconnaissable comme un adaptateur et la consultation de sa source est directe. C'est une fonctionnalité appréciable pour le débogage ou le prototypage. Ceci rejoint le besoin d'utiliser une API existante pour le chargement des méta-modèles (cf. 1.1). L'idéal serait que M-CAST soit une sorte de "plug-in" d'un outil MOF. Il reste à déterminer lequel des outils MOF.

La liste suivante résume les critères de recherche d'un outil MOF dans la perspective du développement et de l'utilisation de M-CAST.

- Éditeur-navigateur de modèles → inspecteur d'objets MOF.
- Plusieurs méta-modèles chargés simultanément.
- Modulaire et ouvert : se servir des fonctionnalités de l'outil à partir d'un langage de programmation, et possibilité d'intégrer de nouveaux modules, de nouvelles fonctionnalités au sein de l'outil (comme M-CAST).

Que ce soit l'utilisation d'une API pour le chargement de méta-modèles ou le prototypage, l'intégration des concepts de type et d'instances dans l'outil MOF est bien sûr souhaitable. Bien évidemment, il sera difficile trouver un outil MOF intégrant nos extensions pour le niveau M-zéro sans que nous n'ayons pas nous-même "prouver" le bien-fondé de celles-ci en les implémentant. Pour cela il est plus logique de ne pas additionner l'intégration des CI et CT à nos critères de recherche et d'envisager plutôt de l'ajouter nous-même à un outil déjà existant.

1.4 L'outillage existant

Pour l'utilisation de l'API de chargement de méta-modèles, seul l'outil M3J offre une possibilité car son code d'implémentation est en accès libre. Il eut donc été une bonne base de départ. Malheureusement, à l'époque où nous avons commencé nos travaux, M3J n'existait pas. *dMOF* était même encore à l'état de prototype avec un grand nombre de caractéristiques MOF non prises en compte.

Nous avions déjà décrit rapidement le fonctionnement typique des outils MOF dans le chapitre 5. Ces outils sont une implémentation "directe" des spécifications du MOF : avec la description UML ou MODL d'un méta-modèle ils génèrent les interfaces IDL correspondantes et le code d'implémentation Java associé. Aucun ne fournit d'inspecteur d'objets MOF[68], d'environnement où il est possible d'instancier un ou plusieurs méta-modèles et de manipuler "graphiquement" les instances.

[67] Car au départ DARE générait une classe Smalltalk par tâche.

[68] Cela reste un projet du DSTC.

Une alternative est d'employer les outils *IdlScript* ou *CorbaWeb* du LIFL [IDL 99][MER 96]. Ces deux outils permettent respectivement de manipuler des objets Corba à partir d'un langage de script et à partir d'interfaces Web. L'idée, ici, serait d'utiliser le langage de script ou les interfaces web pour instancier des méta-modèles, à travers le référentiel Corba généré, et éditer ces instances sans devoir continuellement développer des programmes test, compiler et exécuter. Cette approche correspond à notre besoin d'inspecteur d'objets MOF. Néanmoins, la vue que proposaient ces outils des objets MOF est une vue Corba : pour un objet représentant TPA, le développeur ne voit pas une propriété *sub* mais un ensemble d'opérations `sub`, `add_sub`, ... Une grande part de la sémantique de ces objets MOF est perdue. Nous aurions pu pourtant nous servir de ces outils comme une base pour M-CAST. Un inconvénient est qu'ils nécessitent une génération par un outil MOF des interfaces IDL correspondantes au(x) méta-modèle(s). De plus, nous avions en tête de fournir un outil MOF plus dynamique visant à prototyper les méta-modèles en eux-mêmes. Le fait d'utiliser directement le support Corba (avec IdlScript ou CorbaWeb) vient limiter les possibilités réflexives qui se révèlent nécessaires pour ce type de prototypage.

ex. M1

2 Le prototypage de méta-modèles MOF

Intégrer M-CAST dans un outil MOF implique que celui-ci dispose d'un certain nombre de fonctionnalités. Aucun outil n'est suffisamment fourni. De plus, au travers du développement du système DARE, nous nous sommes aperçus que les outils MOF existants ne facilitent ni l'apprentissage de la méta-modélisation ni le prototypage de méta-modèle MOF.

2.1 Prototyper un référentiel de modèles MOF

Depuis l'adoption de l'architecture MDA par l'OMG, le MOF est sorti de l'anonymat au sein de la communauté "génie logiciel". En plus du manque de clarté vis-à-vis de son utilité (cf. chapitre 5), l'anonymat du MOF a longtemps été alimenté par les bases requises pour sa compréhension :

- la méta-modélisation,
- l'utilisation d'UML pour la méta-modélisation,
- Corba.

La méta-modélisation, même s'il s'agit d'un processus similaire à la modélisation, implique un niveau d'abstraction supplémentaire. Un méta-modèle définit une façon de modéliser des phénomènes alors qu'un modèle définit un type de phénomènes. Un méta-modèle englobe un espace de données plus vaste qu'un modèle. La représentation mentale de cet espace est donc plus difficile. D'autre part, les instances d'un modèle peuvent être qualifiées de "concrètes" alors que celles d'un méta-modèle demeurent abstraites. La méta-modélisation est considérée comme un exercice difficile.

L'emploi d'UML pour créer des méta-modèles a été au départ source de confusions : y avait-il un rapport entre les méta-modèles créés en MOF et les modèles créés en UML ?[69] De plus, un grand nombre de formalismes de méta-modélisation existaient déjà [REV 96] : Entité/Association, Telos, GoPRR, M2SL... Pourquoi encore un nouveau formalisme ?

Corba. Enfin tester méta-modèle, c'est-à-dire créer des instances de celui-ci, nécessite l'utilisation de Corba. Ce dernier n'a pas rencontré le succès commercial et universitaire escompté. Comme le MOF n'était, au départ, qu'un élément de l'architecture OMA, son expansion en a été freinée[70]. L'utilisation de Corba pour instancier un méta-modèle[71] ralentit le prototypage d'un méta-

[69] La réponse est non.

[70] JSR-40 est une des initiatives pour rendre le MOF indépendant de Corba (pour le contexte dynamique) [JSR 02].

[71] Car les outils MOF n'offrent pas la possibilité d'instancier directement un méta-modèle.

modèle : il faut lancer la génération des interfaces IDL, pré-compiler l'IDL, compiler le code d'implémentation, créer un (petit) programme test, le compiler puis exécuter le tout. Si les modèles créés car au programme test ne correspondent pas à ce qu'on attend, il faut modifier le méta-modèle, relancer la génération des interfaces IDL, etc. Le prototypage de méta-modèle peut sembler inutile pourtant il est indispensable. Dans une nouvelle version de DARE [BOU 99], l'entité *Task* hérite de l'entité *Resource*. L'obligation de sa présence s'est révélée au fil des nombreuses créations de modèles de tâches. L'intégration de l'aspect coopératif dans un méta-modèle de workflow (un des objectifs de COW) nécessite-elle aussi la création de modèles pour concevoir des entités cohérentes. ex. M2

Un éditeur de modèles MOF (déjà cité en 1.3) se révèle, dans ce contexte, très bénéfique. La création rapide de modèles permet au concepteur néophyte de "se faire une idée" de son méta-modèle et donc de se familiariser avec la méta-modélisation et le MOF. Ce type d'outil évite aussi au concepteur chevronné la création de programmes "test" lors de la définition d'un méta-modèle. Le prototypage est encore plus pratique si l'éditeur de méta-modèles intègre l'éditeur de modèles, et que chaque modification du méta-modèle se répercute directement sur les instances (i.e. modèles) déjà créées. Le concepteur voit ainsi les conséquences de chaque modification du méta-modèle.

Notre opinion est que les outils MOF actuels bride l'utilisation du MOF. La présence au sein de l'un d'eux d'un réel éditeur (méta-modèles et modèles) avec des caractéristiques réflexives supprimerait ces limites.

2.2 Intégration du niveau M0

Nous avions décrit le niveau M0 dans le chapitre 3 introduit pour diminuer le nombre de liens d'adaptation à définir par le concepteur. Dans le chapitre 5, nous avons intégré le niveau M0 au MOF en définissant des extensions à celui-ci. Ces extensions sont destinées à l'origine pour l'application de CAST sur le support MOF. Mais nous avons indiqué les autres bénéfices de ces extensions : les mécanismes propres au concept d'instance tels que la vérification de typage additionnelle ou la cardinalité secondaire seraient directement générées par un outil MOF (s'il intègre ces extensions). Toutefois, ainsi que nous l'avons dit en 1.3, l'implémentation de ces extensions est à notre charge.

L'intégration du niveau M0 dans l'éditeur MOF "réflexif" constitue un apport considérable. Le concepteur peut enfin, en instanciant les modèles, disposer d'exemples concrets. La modification d'un modèle lui permet de voir les répercussions sur les éléments M0 et donc de parcourir facilement un grand nombre d'éléments M0. La modification du méta-modèle a non seulement une incidence sur les modèles mais aussi sur les éléments M0. Les conséquences d'une modification sont encore plus éloquentes.

Notre motivation, issue des besoins de M-CAST, de développer un outil MOF plus dynamique, a été renforcée par les bénéfices qu'il peut apporter à l'utilisation du MOF elle-même.

3 Description de RAM3

Devant les carences précédentes, nous avons développé un outil MOF dynamique (intégrant un éditeur), appelé RAM3 (RApid Manipulation of MOF Metadata), dont les objectifs sont de faciliter la prise en compte du niveau M0, le prototypage de méta-modèle MOF et de servir de base pour la réalisation de M-CAST.

Dans cette partie, nous décrivons les différentes caractéristiques de RAM3 au travers d'exemples d'utilisation : prototypage, outil de modélisation basique, développement MOF, transformation. Nous dressons, à la fin de cette partie, une liste des caractéristiques de RAM3 pour en donner une vue générale.

3.1 Prototypage

Nous prenons la situation où nous avons défini le méta-modèle DARE avec RAM3 et nous voulons ajouter une association d'héritage sur l'entité *Role*. La saisie d'un méta-modèle est détaillée dans la documentation (annexe B) qui décrit, en autre, les détails de la saisie d'un méta-modèle.

ex. M2

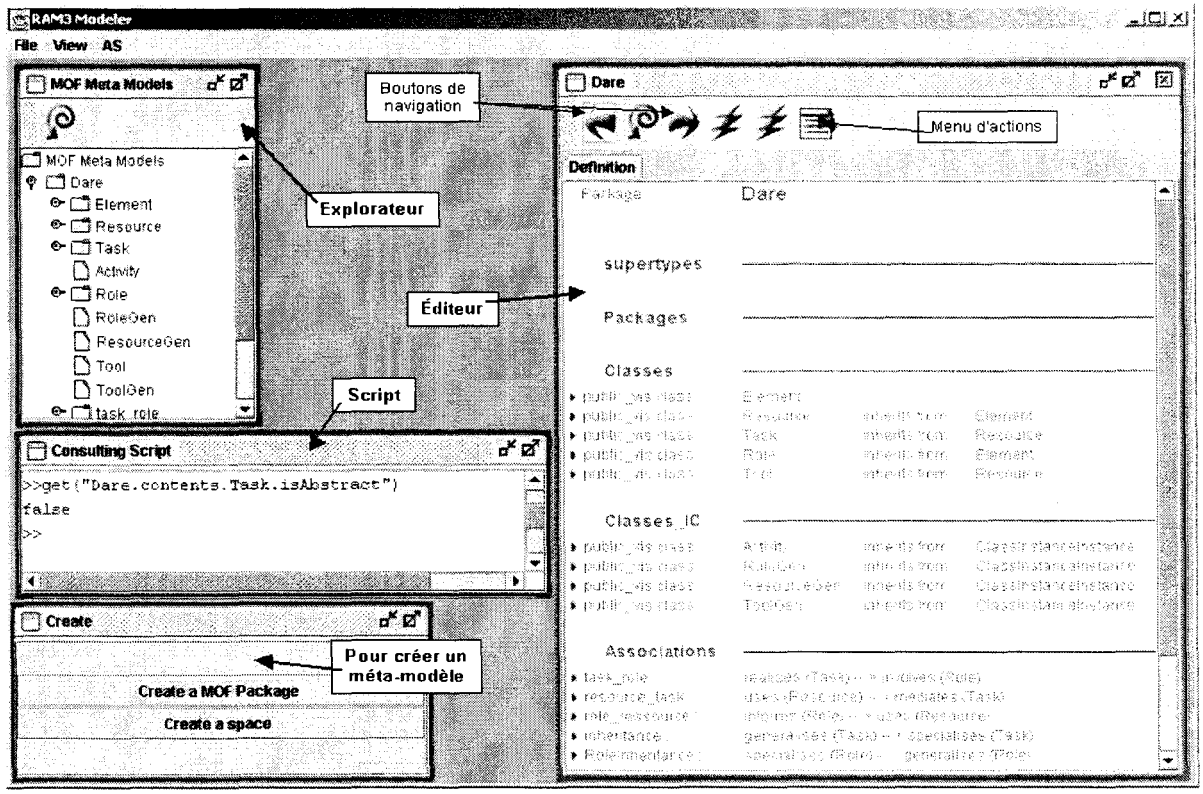
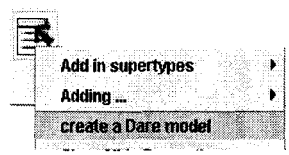


figure 73. Environnement de RAM3

La figure 73 montre l'environnement de RAM3. Il est constitué de trois types de composants :

- Des explorateurs : arborescences d'objets MOF,
- Des éditeurs : affichent toutes les propriétés d'un objet MOF,
- Des fenêtres de script : permettent de manipuler les objets MOF à partir d'instructions Java.

L'éditeur de la figure 73 affiche les propriétés du paquetage DARE. Celui-ci est identique au méta-modèle décrit dans le chapitre 3. Les concepts de type sont rangés dans la partie *Classes* et les concepts d'instance dans la partie *Classes_IC*. A partir de ce méta-modèle, il est possible d'en créer des instances. La figure 74 représente le modèle *HLC* contenant *FormationBase* (du chapitre 3). La valeur de chaque attribut est modifiable tout comme l'ensemble des objets d'une référence (la figure 74 montre l'édition possible du nom de la tâche *Forum* et l'ajout d'un rôle dans sa référence *involves*). En plus de proposer la création directe d'instance de méta-modèles MOF, RAM3 permet d'instancier les modèles MOF. Cette double instanciation suit les extensions du MOF présentées dans le chapitre 5.



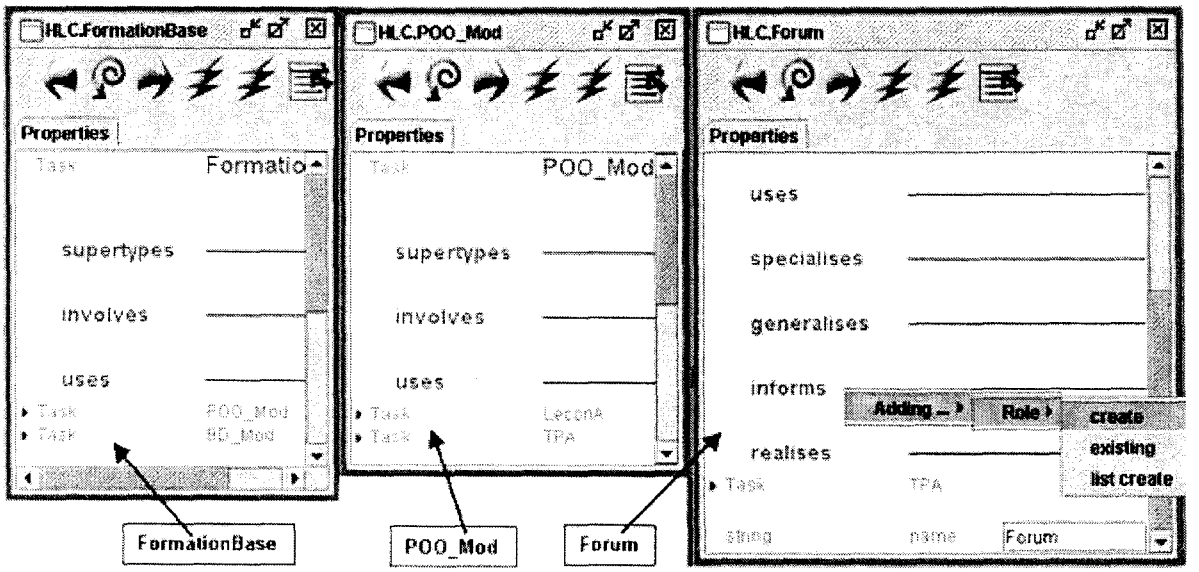


figure 74. Modèle d'HLC

La figure 75 montre une instance du modèle *HLC* : *Promo01_02*. Cette dernière permet de créer des instances de chaque élément d'*HLC*. Sur cette même figure apparaît l'activité *FormaBase01_02*, une instance de la tâche *FormationBase*. Dans celui-ci, apparaît les points d'entrées secondaires : par exemple, pour *sub*, une liste des *POO_Mod*, une liste des *BD_Mod*. Il est possible de n'autoriser qu'un seul module de POO. Pour cela, il faut utiliser les cardinalités secondaires. La figure 76 montre l'association de la cardinalité 0..1 à *POO_Mod*. Automatiquement, *FormaBase01_02* ne gère plus qu'un seul module de POO.

ex. M0

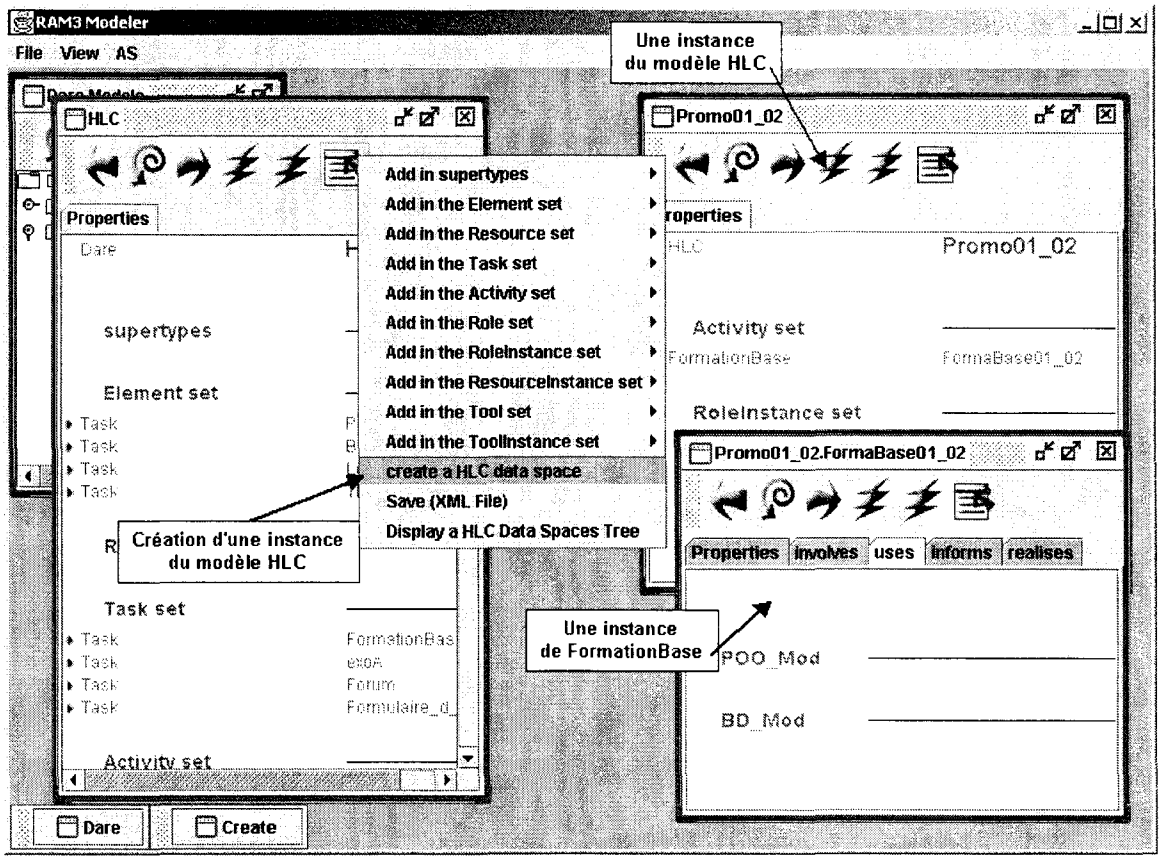


figure 75. Des éléments M-zéro pour HLC dans RAM3

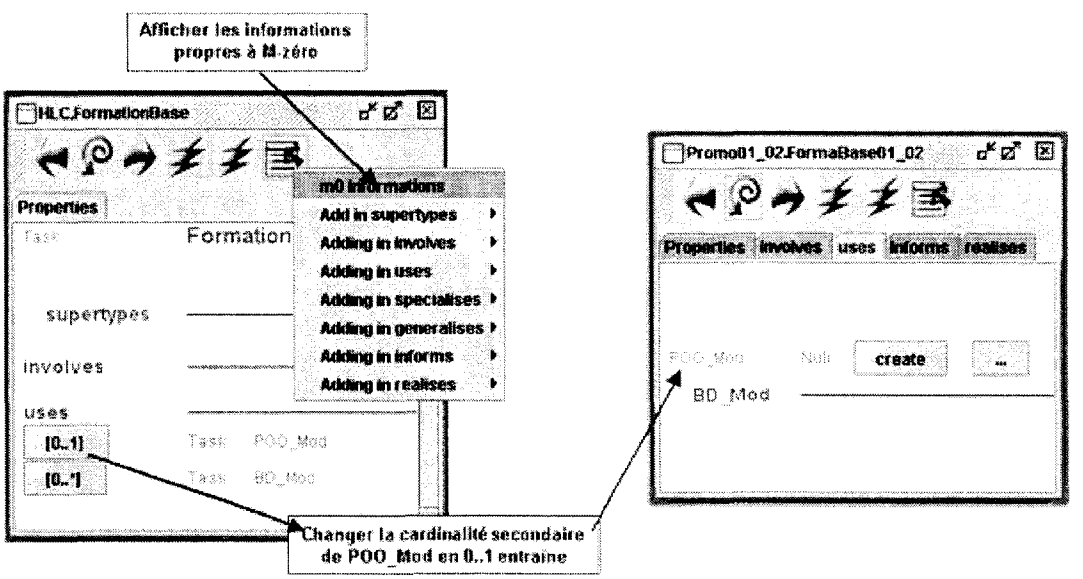
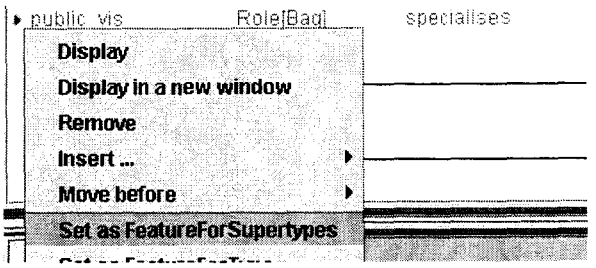


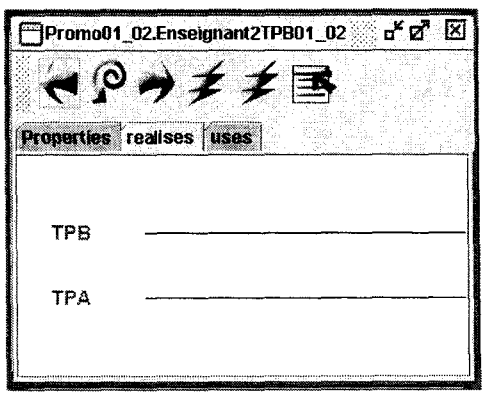
figure 76. Modification de la cardinalité secondaire et répercussion

Dans ce contexte de "prototypage" de modèles, il est possible d'imaginer la suppression de *TPA* pour *FormationBase*, ce qui supprimerait les éléments *TPA* de *FormaBase01_02*. Si un *TPAbis* était ajouté à *FormationBase*, une liste (vide) de *TPAbis* apparaîtrait dans l'éditeur de *FormaBase01_02*.

L'objectif est maintenant de tester les bénéfices d'une association d'héritage entre les rôles. Pour cela, on utilise comme repères le rôle *Enseignant* de *TPA* et son instance *EnseignantTPA01_02* de *TPA01_02*. L'association d'héritage sur *Role* est créée (*RoleInheritance* : *specialises* → *generalises*) en indiquant que *specialises* est le *featureForSupertypes*.



Un nouveau type de rôle *Enseignant2* est référencé par une nouvelle tâche *TPB*. *Enseignant2* hérite de *Enseignant* (de *TPA*). Sont créés finalement l'activité *TPB01_02* (instance de *TPB*) et le *Enseignant2TPB01_02* (instance d'*Enseignant2*). L'affichage de ce dernier nous indique sur l'apport de l'héritage : une personne qui est enseignant dans un TPB peut intervenir dans des *TPA* et des *TPB*. Si cet apport est jugé inutile, il suffit de supprimer l'association d'héritage. Le point d'entrée secondaire TPA disparaît alors de *Enseignant2TPB01_02*.



Les caractéristiques de RAM3 que nous avons présentées sont les suivantes :

- instanciation de méta-modèles
- instanciation de modèles
- édition des deux types d'instances (et des méta-modèles)
- intégration des extensions M0 (vérification de typage additionnelle, cardinalité secondaire, points d'entrée secondaires, héritage).
- mécanismes réflexifs

3.2 Un début d'outil de modélisation

Le prototypage d'un méta-modèle n'est pas obligatoirement réalisé par le concepteur uniquement. Les utilisateurs finaux peuvent, eux aussi, faire parti des juges de l'acceptation du méta-modèle. Dans le cas de DARE, les entités du méta-modèle ont pour objectif d'être forts proches de concepts "familiers" et "intuitifs" pour être ainsi facilement compréhensibles par les utilisateurs finaux. De cette manière, les utilisateurs peuvent eux-mêmes modéliser leurs propres activités. Dans ce type de contexte, les utilisateurs interviennent dans le prototypage.

Les interfaces génériques de RAM3 sont peut-être accessibles pour un informaticien (aguerri), elles le sont sûrement moins pour les non-informaticiens. Si, pour évaluer la facilité du méta-modèle, on demande à un utilisateur de modéliser une activité (i.e définir une tâche) avec les interfaces graphiques précédentes, il risque d'être déboussolé par le caractère "brutes" de celles-ci[72]. Pour atténuer cette froideur, les fonctions d'affichage sont personnalisables. Quatre fonctions sont modifiables :

[72] Les problèmes d'utilisabilité n'ont pas été, jusqu'à présent, une de nos préoccupations majeures. Néanmoins l'équipe NOCE possède des compétences très fortes en IHM qui seront mobilisées dans l'avenir.

- Affichage complet : quand l'objet est la source de l'éditeur.
- Affichage raccourci : lorsque l'objet apparaît dans un liste d'objets.
- Édition : pour la modification d'une valeur (essentiellement les types de bases).
- Instanciation : fonction invoquée quand une instance de l'objet est créée. Elle peut consister en une suite de question pour définir les propriétés principales de ce type d'instances.

Ces fonctions sont modifiables drectement dans RAM3 pour chaque type d'élément. Pour l'instant ces fonctions sont à implémenter en Java avec une API d'accès simplifié aux méta-modèles, modèles et éléments Mézéro présents dans RAM3. Le canevas d'affichage dispose lui aussi d'un certain nombre de fonctions, comme la gestion d'une barre d'outil, d'un menu d'action ou l'ajout de liens, pour faciliter l'implémentation de ces fonctions d'affichage. L'implémentation peut être modifiée indéfiniment sans devoir quitter RAM3 ou recharger le méta-modèle DARE. La figure 78 montre la modification de l'affichage complet associé à *Task* (le code n'est pas complet) la figure 77 illustre le nouvel affichage.

ex. M2

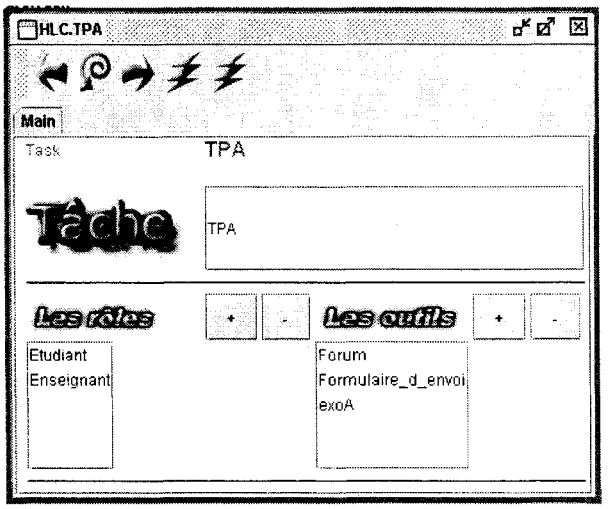


figure 77. Le nouvel affichage des tâches

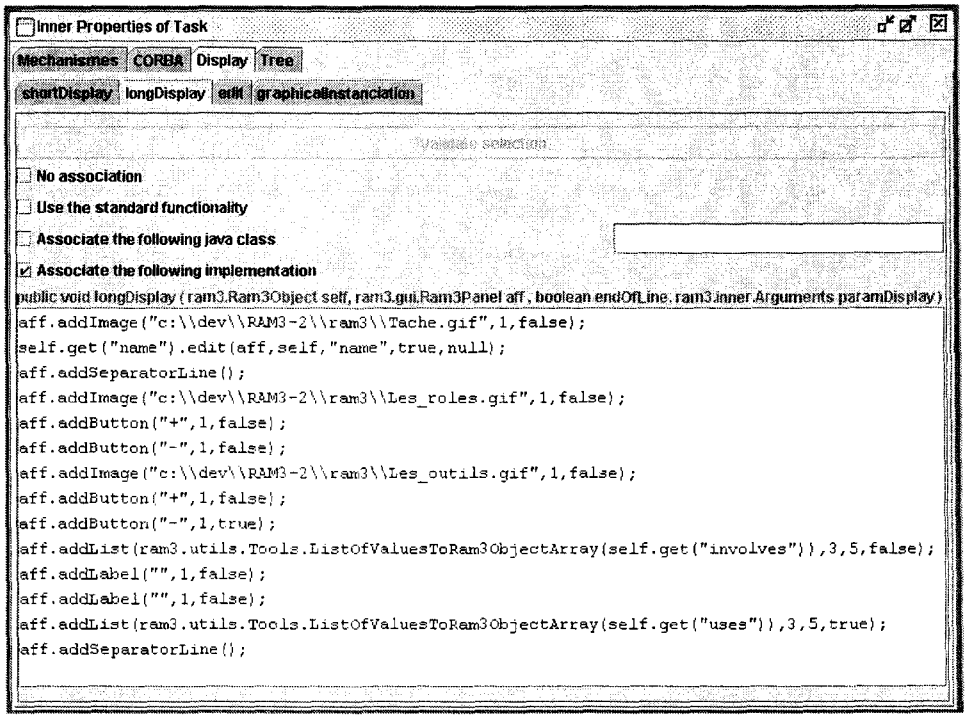


figure 78. Modification de la fonction d'affichage complet de *Task*

Cette fonctionnalité de RAM3 permet à l'affichage de focaliser sur certaines propriétés (et d'en cacher d'autres) pour rendre plus aisée l'édition de modèles et ainsi être plus proche des utilisateurs.

Caractéristiques supplémentaires de RAM3 présentées :

- Affichage des éléments paramétrables
- API Java d'accès aux éléments de RAM3

3.3 Outil de développement MOF

Le "développement" de certains méta-modèles MOF nécessite un codage important. Il s'agit par exemple d'exportation MOF : un référentiel de modèles non MOF existe déjà et le concepteur désire associer à celui-ci un accès normalisé (i.e. MOF). La situation souhaitée est de disposer des objets CORBA/MOF qui permettent d'accéder aux modèles contenus dans le référentiel déjà existant. La figure 79 illustre un exemple de ce type où le référentiel serait construit à l'aide d'une base de données.

Dans ce type de situations, le méta-modèle MOF d'exportation nécessite une phase de codage importante : chaque accès à une caractéristique d'un élément de modèle sollicite un accès à la base de données. Si le concepteur utilise un des outils MOF existants, il définit d'abord le méta-modèle dans l'outil, en génère les interfaces IDL, puis implémente celles-ci dans le langage où il dispose de l'API pour accéder à la base de données du système. Le langage Java avec son paquetage JDBC peut suffire dans grands nombres de cas. Ensuite vient la phase de test avec ses innombrables modification/compilation/exécution.

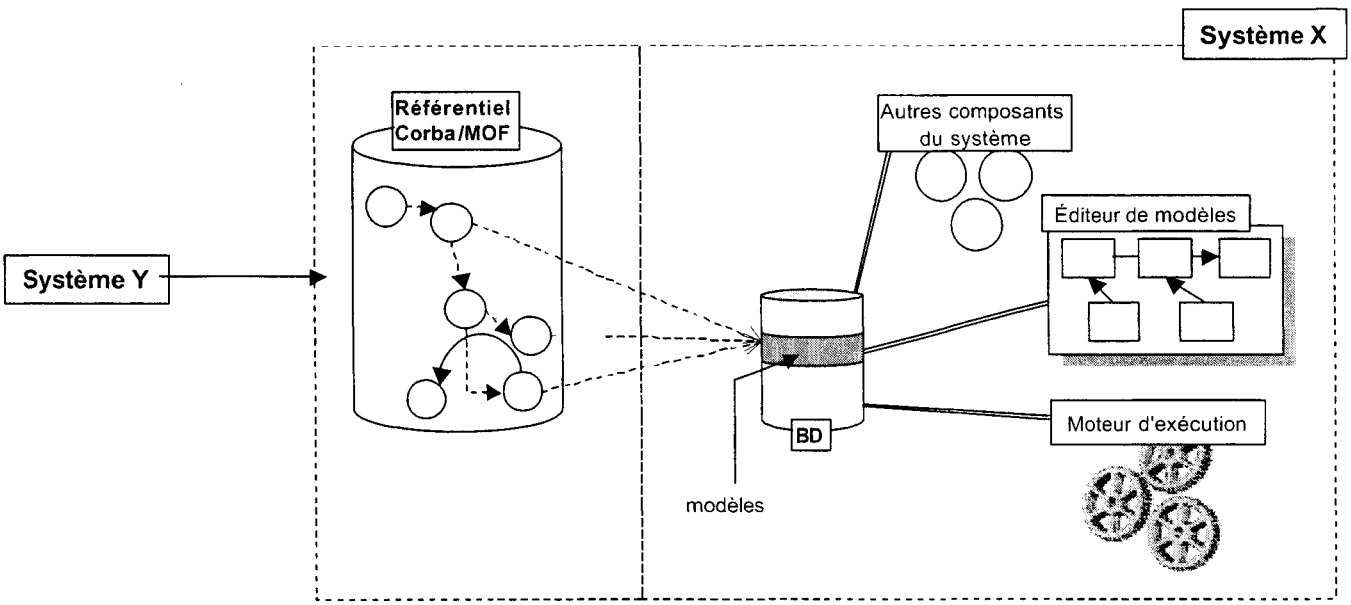


figure 79. Exemple d'exportation MOF d'un référentiel de modèles non MOF

Avec RAM3, nous avons décidé de supprimer la dissociation méta-modèle/implémentation. L'implémentation des opérations, des méthodes d'accès aux attributs ou aux références peut être saisie directement dans RAM3. Pour ces dernières méthodes, il suffit, par exemple, de créer une opération `X()` ou `set_X()` afin d'implémenter la consultation ou l'affectation de la caractéristique *X*. L'implémentation peut être, là aussi, modifiée indéfiniment. La figure 80 montre l'implémentation de la méthode de consultation pour la référence *uses*.

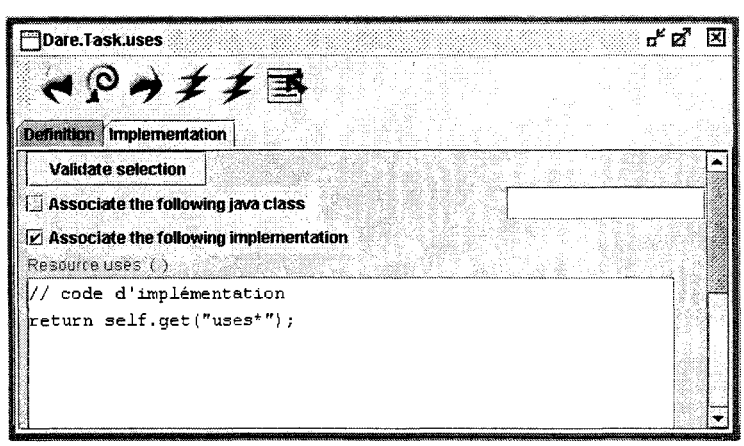


figure 80. Méthode d'accès de type *get* pour la référence *uses*

Remarque : dans cette figure le nom de la caractéristique à consulter est suivie d'une étoile. `self.get("uses*")` *signifie ne pas utiliser la méthode de consultation et retourner la véritable valeur.*

L'association du code au méta-modèle au sein de RAM3 et la possibilité de changer dynamiquement l'implémentation (même si des "instances" existent) accélèrent le prototypage de méta-modèles d'exportation. Il est même possible de modifier le comportement de la gestion des points d'entrées. Par exemple, si chaque affectation de valeur aux propriétés des tâches doit générer un événement, il est possible de modifier directement la *méthode "interne" set* de *Task*. Cette méthode centralise tous les accès de type *set* pour les tâches (cf. figure 81).

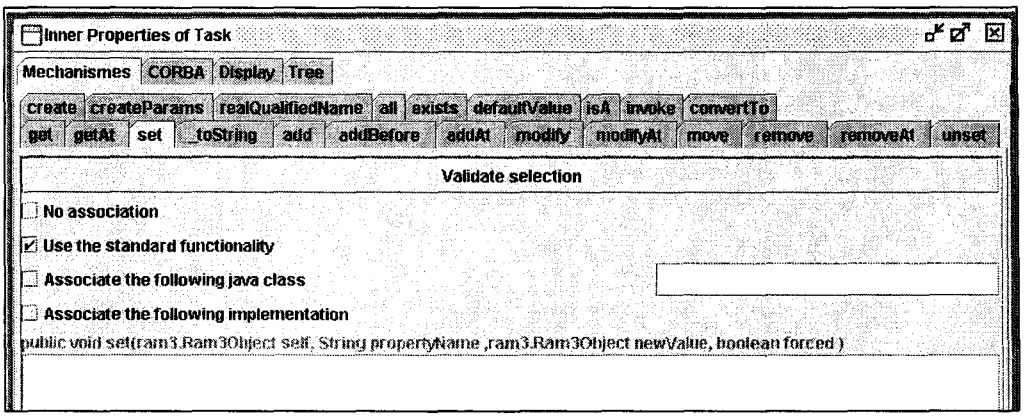


figure 81. Modification des mécanismes internes des tâches

Caractéristiques supplémentaires de RAM3 présentées :

- Implémentation des opérations et des méthodes d'accès aux attributs et aux références.
- Modification de la gestion des points d'entrée : changer l'implémentation du get, set, add, modify, invoke, ...
- Implémentation modifiable dynamiquement

3.4 Des dispositions à la transformation de modèles

La transformation de modèles, dans son sens commun [LEM 98][REV 01], ne signifie pas modifier un modèle mais plutôt translater le modèle dans un autre contexte, c'est-à-dire créer une copie de ce modèle, exprimée dans un autre méta-modèle. Dans RAM3, il est possible de charger plusieurs méta-modèles simultanément. Il est alors facile de créer un méta-modèle factice qui ne contient qu'une entité dont les opérations correspondraient à des fonctions de transformation de modèles d'un méta-modèle A à un méta-modèle B. Néanmoins, pour la transformation de modèles, nous avons préféré ajouter la possibilité de créer des classes Java à l'intérieur de RAM3. Il y a trois raisons à cela :

- L'utilisation d'un méta-modèle factice implique que le "transformateur" soit un Ram3Objet. Chaque invocation d'opération passe par le mécanisme "interne" d'invocation de l'objet RAM3. L'exécution du transformateur est fortement ralentie.
- La classe Java peut plus facilement être utilisée en dehors de RAM3.
- La structure d'une classe Java est plus familière pour le développeur.

Le principal problème est qu'une classe Java chargée dans la machine virtuelle ne peut plus être modifiée (dans la version standard de *Java2*). À cause de ce manque de réflexivité du langage Java, nous avons ajouté un éditeur de classe Java sans nom de classe. Ce dernier est changé à chaque modification de la classe. La figure 82 montre un exemple de classe éditée.

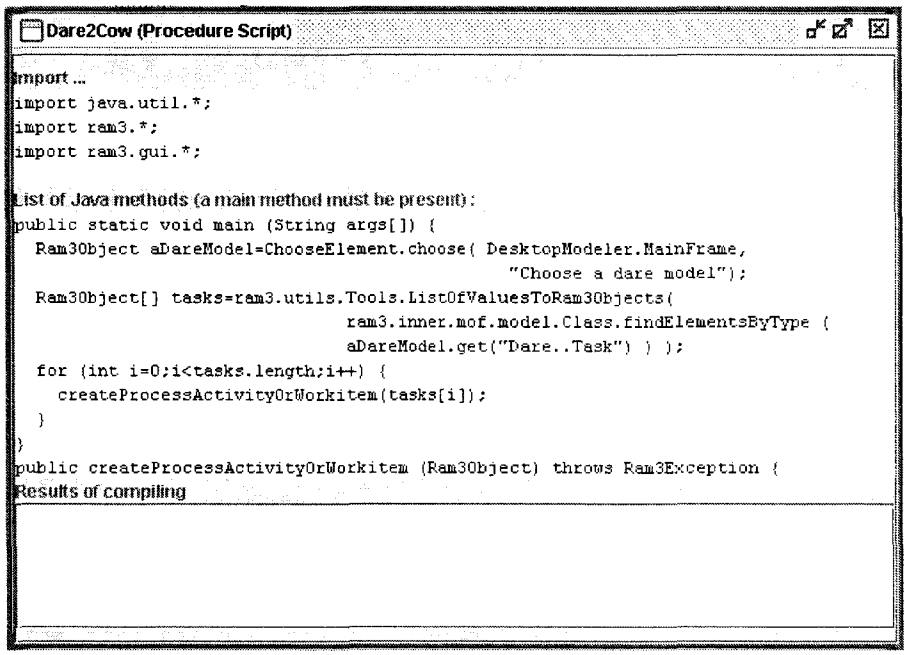


figure 82. Une classe Java au sein de Ram3

3.5 Un serveur de méta-objets MOF en Java

Certains systèmes n'ont pas besoin d'avoir un référentiel de modèles accessible à distance ou certains concepteur souhaitent développer un référentiel de méta-données sans support Corba (MOF ou non). Pour cette raison, nous avons envisagé la possibilité d'utiliser RAM3 comme un moteur d'exécution sans se préoccuper du support Corba. RAM3 peut être exécuté sans interface graphique et/ou sans support Corba (cf. listing 1). Il devient dans ce cas un simple serveur de méta-objets MOF en Java. Notre idée est de proposer, dans cette perspective d'utilisation, un moteur d'exécution pour tout système flexible à trois niveaux. Même si l'objectif de RAM3 reste le prototypage, nous avons remarqué que sur une machine suffisamment puissante (PC - Pentium III 1Ghz – RAM 256 Mo) l'exécution de RAM3 ne souffre pas de lenteur[73]. Les avantages d'un tel référentiel de modèles (par rapport à ceux générés par dMOF ou M3J) sont

- La possibilité de modifier dynamiquement le méta-modèle,
- La possibilité de lancer à tout moment l'interface graphique de RAM3 (par exemple, pour des besoins de monitoring).
- La présence d'un support pour le niveau M-zéro.
- L'utilisation possible par l'implémentation du système, utilisant RAM3 pour son référentiel de modèles, de l'API de RAM3 vis-à-vis de l'exécution dynamique de code Java (par exemple pour la création d'un shell).

[73] Pas de mesure pour l'instant.

- Aucune présence du support Corba.

Des tests de performance feront partie de nos objectifs futurs.

```
ram3.Init.execute(args,false,false); // lancement de RAM3
Ram3Object metamodel=ram3.utils.XmlParser("DARE.m2.xml");
Ram3Object a_model=ram3.utils.XmlParser("HLC.m1.xml");
Ram3Object contents=A_model.get("contents");
```

listing 1. Utilisation de RAM3 dans un programme Java

Le fait de pouvoir exécuter RAM3 sans interface graphique et sans support Corba est aussi motivé par notre volonté de proposer d'autres supports d'implémentation au MOF. La projection de méta-modèles MOF vers le support WSDL/SOAP (implémentée grâce à RAM3) a déjà été amorcée lors d'un stage[74]. Le support EJB est aussi très séduisant de par sa présence plus importante dans le monde industriel. Il fait aussi partie de nos projets. Néanmoins nous avons commencé par WSDL/SOAP car l'emballage/déballage de requêtes distantes est modifiable (ce qui n'est pas le cas du support EJB). Ceci permet à RAM3 de recevoir une requête WSDL et d'effectuer le traitement correspondant au contenu de la requête (technique similaire à celle utilisée avec le DSI de Corba).

4 Les sources d'inspiration

Nos sources d'inspiration sont liées à trois domaines : celui des interfaces ou éditeurs, celui des architectures de système et le domaine du prototypage.

Interfaces/éditeurs. Les interfaces graphiques d'édition de méta-objets MOF sont bien sûr inspirées des outils de meta-CASE [MET 96]. Ces outils ont pour objectif de permettre au développeur de construire son propre environnement de conception. Cette personnalisation (ou malléabilité) peut se faire de plusieurs façons (la classification de Morch - cf. chapitre 3 – est toujours valable) : paramétrage, composition, langage personnel. DOME [DOM 02] propose une vingtaine de méthode de modélisation existantes et la possibilité de créer des artefacts graphiques correspondant à des composants personnels. Des outils comme METAGEN [REV 95] ou SEMANTOR [LEM 00] permettent quant à eux de créer son propre méta-modèle et de développer des applications à partir de celui-ci. Objecteering [OBJ 02] ou ObjectMaker [OBJ 02] propose quant à eux la possibilité d'étendre des méta-modèles existants comme UML (concept de profile UML [BOR 00] pour Objecteering). C'est l'outil SEMANTOR et plus précisément le s-Browser (ancêtre du premier) [BEZ 98] qui nous a inspiré lors de la conception des interfaces graphiques de RAM3. Le sBrowser est issu des travaux sur les sNets [BEZ 98] de Bézivin. Comme le montre la figure 83, il permet de naviguer sur tous les éléments quelque soit leur degré d'abstraction. C'est cette idée de transversalité qui est à la base des éditeurs de RAM3. Les formats d'affichage et la personnalité de ces derniers sont par contre plus issus d'une volonté de personnalisation/malléabilité (issues de l'expérience de l'équipe NOCE) que de travaux précités. Les raisons du choix d'une modélisation à partir d'éditeur/navigateur plutôt qu'une modélisation à la UML est abordée dans le chapitre 7.

[74] Le stage avait pour intitulé : "Création d'un support WSDL pour le standard de méta-données MOF". Il s'est effectué sur la période Mars à Mai 2002 avec deux étudiants du DESS MICE. Les règles de projection vers le langage WSDL ne sont faite qu'à 30 %. Un module a été rajouté à RAM3 pour générer les interfaces WSDL correspondantes à un paquetage choisi par l'utilisateur.

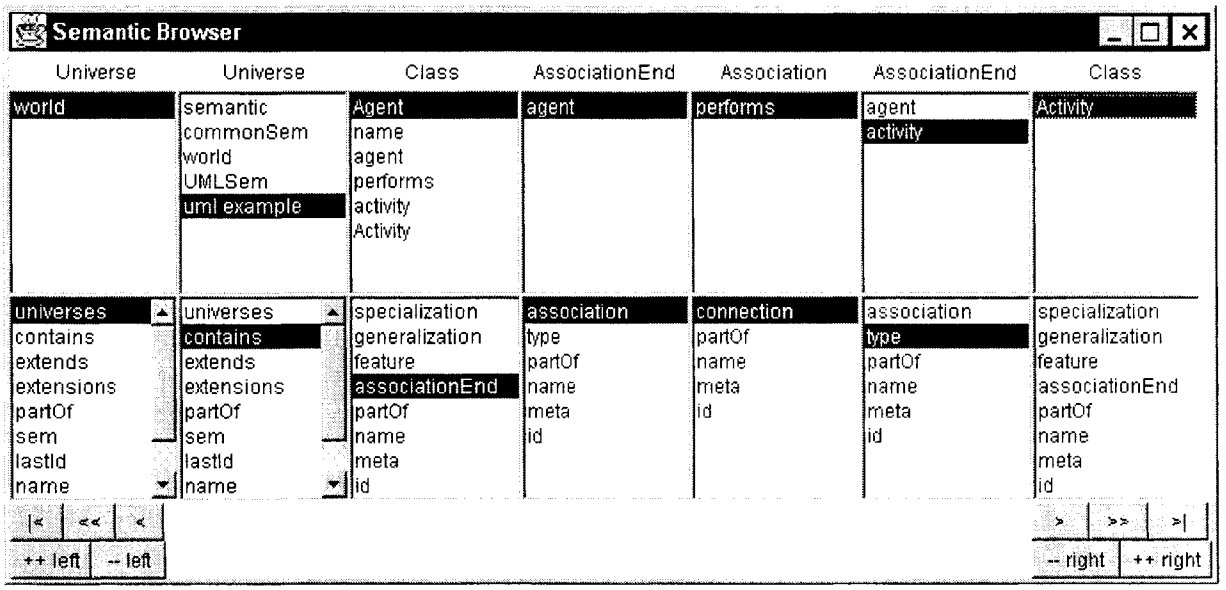


figure 83. Le sBrowser (sNets)

L'architecture interne de RAM3 suit le principe des extensions M-zéro du MOF (cf. chapitre 5) : à chaque objet de RAM3 est associé un type réel et un type de définition à chaque objet de RAM3. C'est principalement l'approche par méta-objet [MAE 87] qui nous a servi pour la conception de la classe Ram3Object.

Une orientation vers le prototypage. Les perspectives d'utilisation des outils IdlScript et CorbaWeb sont en partie à l'origine des choix d'orientation de RAM3 vis-à-vis du prototypage : prototypage et monitoring d'applications réparties à l'aide d'un langage de script ou d'un navigateur web. Les outils UML dynamiques comme UMLaut [HO 99], ou Objecteering [OBU 02] sont apparus (ou sont devenus suffisamment avancés) que pendant la conception de RAM3.

5 Description de l'architecture de RAM3 *

Cette partie a pour objectif de décrire l'architecture de RAM3. Celui-ci a été entièrement développé en langage Java.

5.1 Aperçu de l'architecture logicielle de RAM3

Le centre de l'architecture de RAM3 est le `Ram3Object`. Il s'agit d'une classe Java où tous les éléments manipulés au sein de RAM3 (méta-modèles, modèles, éléments M0, entiers, chaînes de caractères, ...) sont des instances de cette classe. La figure 84 montre les liens entre les différents paquetages de RAM3 et la classe `Ram3Object`. Chacun des cinq éléments est détaillé dans des sections ultérieures. Les attributs de la classe `Ram3Object` (référencés sur la figure comme la structure) gèrent, pour une instance, les valeurs qu'elle contient, une référence sur son type, etc. Les opérations du `Ram3Object` sont principalement les méthodes d'accès aux points d'entrée RAM3 (`get`,`set`, ...), les méthodes d'affichage et les méthodes de gestion de l'accès Corba.

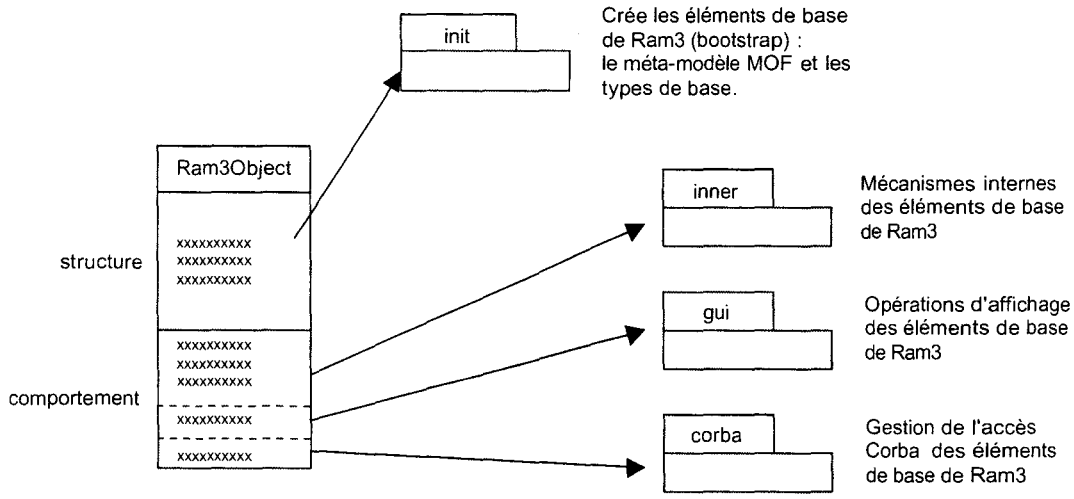


figure 84. Principaux éléments de l'architecture de RAM3

Lorsque RAM3 démarre, il contient un noyau d'éléments : le méta-méta-modèle MOF et les types de bases (entier, flottant, ... correspondant aux types Corba). La création de cet ensemble de `Ram3Object` est l'objectif du paquetage `init`, qui correspond au *bootstrap* de RAM3. Le comportement du noyau et les méthodes d'accès aux points d'entrée RAM3 de celui-ci sont fixés par le paquetage `inner`. Par la suite, nous désignons par *opérations internes* ces méthodes (accès et comportement). L'implémentation des différents formats d'affichage (celui de *Package*, *Class* et *Class_IC* ont été abordés dans la section 3) provient du paquetage `gui`. Enfin l'accès Corba (comme la réception d'une invocation), évoqué ultérieurement, est géré par le paquetage `corba`.

5.2 Ram3Object

La classe `Ram3Object` définit un ensemble d'attributs avec pour chacun un objectif bien défini. La figure 85 fournit une illustration de l'agencement des `Ram3Object`. Il s'agit du cas où RAM3 édite le méta-modèle de DARE. Pour des raisons de clarté, tous les éléments ne sont pas détaillés, seul l'objet représentant l'entité *Task* est complètement décrit.

ex. M2

La classe `Ram3Object` définit l'unique élément d'une architecture où "tout est objet" (*un objet est une instance de classe qui est elle-même un objet ...*). Tous les attributs que déclare la classe sont les propriétés de cet unique concept *Ram3Object*. C'est à partir de ce dernier que vient se construire tous les éléments du MOF. Comme le montre la figure 85, les constructions sont complexes. Mais cette complexité est heureusement cachée par les opérations internes des méta-entités.

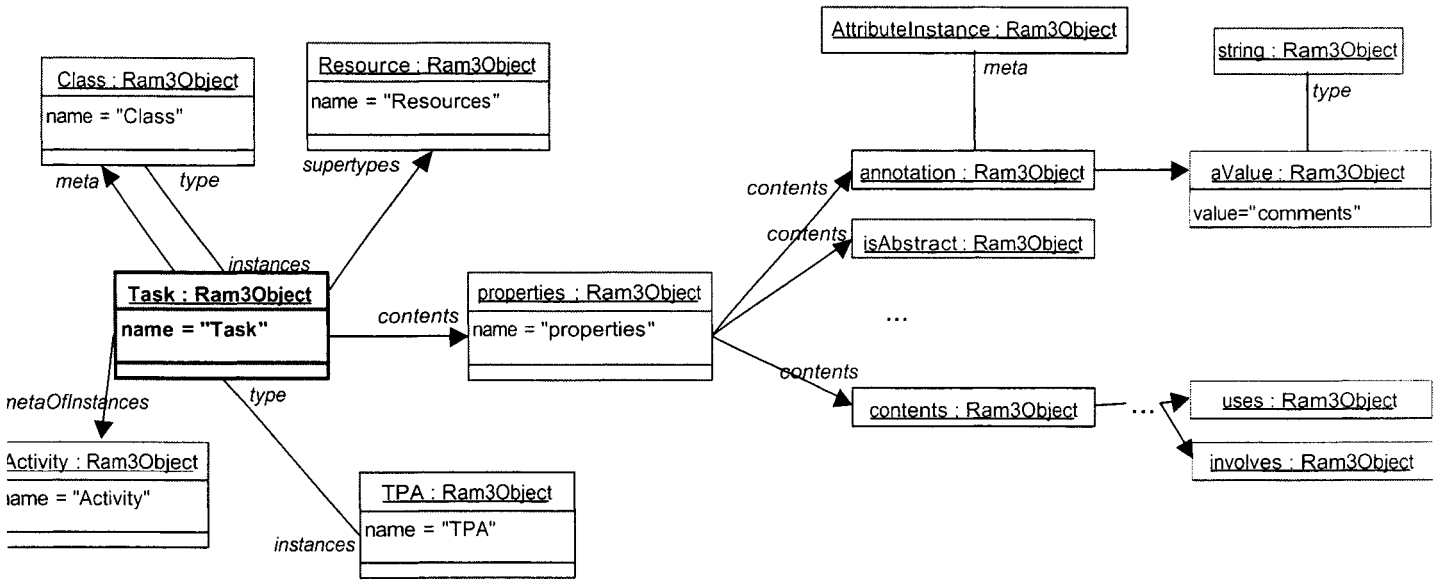


figure 85. Le concept *Task* sous forme de Ram3Object

Nous décrivons maintenant la liste des attributs.

Meta & Type. Le lien *meta* référence le méta-objet. Le méta-objet correspond au type réel énoncé dans les chapitres 3 et 5. Pour qu'un élément M-zéro référence x pour une caractéristique c, il faut le méta-objet de x soit égale au type primaire de c. Ce sont les opérations internes du méta-objet qui sont invoquées. Le lien *type* correspond au type de définition (défini aussi dans les chapitres 3 et 5). C'est là que se trouve la structure secondaire de l'objet.

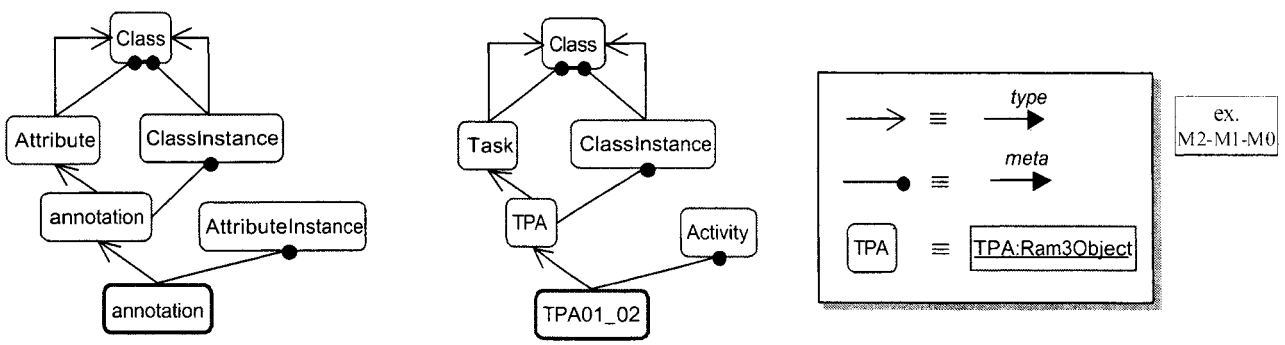


figure 86. Types et métas

Le type et le méta-objet sont identiques pour les entités. Pour la majeure partie des autres cas, ils sont différents. La différenciation est obligatoire principalement pour les extensions M0 (cf. figure 86). Elle apparaît aussi pour les éléments M1 (notamment avec *ClassInstance*) mais elle reste moins nécessaire. Les attributs d'une classe MOF nécessitent par contre cette différenciation. La définition d'un attribut au sein d'une classe devient une propriété pour chacune de ses instances. Ces propriétés sont, pour RAM3, un *AttributeInstance* (cf. figure 85 et 14) et contiennent une valeur. Le fait de

contenir une valeur n'est pas défini par l'attribut mais par la structure primaire, ici *AttributeInstance*. L'attribut ne définit que le type de la valeur. La différenciation est bien obligatoire[75].

ex. M1

L'instanciation a un schéma particulier. Il s'agit d'abord d'une opération invoquée (`newInstance`) sur un *créateur*, qui peut être une classe MOF (e.g. *Task*) ou une instance de classe (e.g. *TPA*). Cette opération crée un objet dont le type est le créateur. Le méta-objet de la nouvelle instance est la valeur de l'attribut Java `metaOfInstances` du type du créateur[76]. Pour un concept de type, cette propriété indique quel sera le méta-objet des éléments M0 issus de ce CT. Elle correspond à la référence *realisedBy* de l'association *realises* des extensions M-zéro. L'opération interne `create` du méta-objet de la nouvelle instance est ensuite invoquée. Lorsqu'une classe est instanciée, c'est l'opération `create` de *ClassInstance*, méta-objet de chaque entité (dans RAM3), qui est invoquée. Elle parcours la classe à la recherche de constructeurs (opérations du même nom que la classe) à invoquer. L'attribut **instances** de Ram3Object mémorise toutes les instances d'un objet.

ex. M2

Contents & value : les propriétés d'une classe, comme par exemple la structure et le comportement qu'elle définit[77], sont stockées grâce à l'attribut `contents`. C'est lui qui contient les valeurs. `Contents` permet à un `Ram3Object` de contenir un ensemble de `Ram3Object`. `Value` sert exclusivement aux valeurs de base. Par exemple, dans RAM3, un entier est aussi un Ram3Object. Son type est *Core.Integer* et il contient, à travers `value`, un entier Java.

Les sous-types & super-types. Les super-types (et les sous-types) sont directement accessibles grâce à l'attribut `supertypes` (`subtypes`). Il y a trois raisons à cela :

- Chacune des opérations internes n'est pas définie pour chaque "type", elle peut être définie dans un des super-types. S'il faut passer par un `get("supertypes")` pour avoir les super-types, alors que `get` regarde dans les super-types où se trouve une opération interne *get* implémentée, il y a une boucle fermée.
- L'héritage doit faire partie du noyau "objet" de RAM3.
- Les liens d'héritage ne sont pas propres qu'aux classes, ils peuvent être aussi présents pour leurs instances. Les gérer en interne est un gain de temps.

`Supertypes` et `subtypes` sont implémentés de manière à ce que l'ajout de x comme super-types de y implique l'ajout de y comme sous-type de x.

Si le **nom** était géré comme une propriété, lorsqu'on souhaite accéder à la propriété x d'un objet, il faut parcourir toutes les propriétés de l'objet et prendre la valeur de la propriété dont le nom est x. Demander le nom d'une propriété revient à demander la propriété *nom* de la propriété. Cette dernière opération entraîne un nouveau parcours de toutes les propriétés de la propriété en demandant le nom de chacune d'elles ... Pour éviter cette boucle fermée, le nom est accessible directement à travers l'attribut `name`.

Tout Ram3Object est contenu dans un autre Ram3Object. Pour indiquer le conteneur auquel appartient l'objet, on utilise l'attribut Java **container**. C'est parce qu'il est l'inverse de `contents` que `container` apparaît dans Ram3Object.

Les opérations internes, les méthodes d'affichage et d'accès Corba sont d'abord des méthodes Java de la classe `Ram3Object`. Chacune de ces méthodes est une redirection vers un objet qui

---

[75] De toute façon, une propriété est un élément M-zéro : le concept *Attribute* crée un attribut qui crée une propriété.

[76] En fait si le *metaOfInstances* du type du créateur est nul, c'est le *metaOfInstances* du méta-objet qui est sélectionné. Si la valeur sélectionnée est encore nulle, alors l'objet n'est pas instanciable. Le *metaOfInstances* du méta-objet est généralement *ClassInstanceInstace* qui est le concept d'instance de base (non modifiable).

[77] La définition de la structure et du comportement se font à travers la référence *contents* de la méta-entité *NameSpace* dont hérite *Class*. Pour cette raison nous avons aussi appelé l'attribut du Ram3Object qui "contient" d'autres Ram3Objects *contents*.

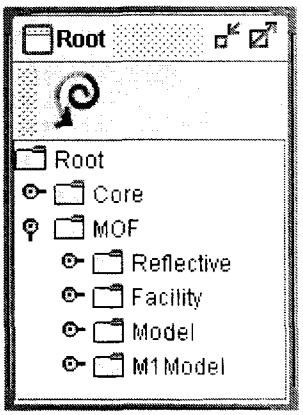
contient le traitement à effectuer, spécifique à l'objet (comme pour CAST). L'attribut **inners** contient pour chacune des méthodes précédentes l'objet traitement correspondant (s'il existe).

5.3 Les éléments du noyau

Au démarrage, se construit un ensemble d'éléments qui constitue le cœur de RAM3. Ce sont ces éléments qui font de cet outil un modeleur MOF. Sans eux, RAM3 n'est qu'un éditeur de Ram3Object. Le premier module créé est **Core**. Il contient tous les types de bases IDL (ceux utilisés par le MOF sont ceux du langage IDL) : `alias`, `any`, `boolean`, `char`, `double`, `enumeration`, `field`, `float`, `long`, `octet`, `sequence`, `short`, `string`, `structure`, `typecode`, `ulong`, `ushort`. Le type `sequence` n'est utile que pour la projection IDL. C'est le type `Core.ListOfValues` qui est utilisé pour les listes d'objets. Le type `space`, qui hérite de `ListOfValues`, introduit le concept d'espace de nommage (recherche par nom possible - les éléments contenu dans un espace, référence celui-ci comme leur conteneur).

Le second module (**MOF**) est l'ensemble des paquetages MOF : `Model`, `Reflective`, `Facility`. Ils sont la complète retranscription du MOF étendu dans RAM3. Les extensions sont en grande partie définies dans un module spécifique : `M1Model`. Il contient la classe `ClassInstanceInstance` dont héritent tous les concepts d'instance. Elle définit le comportement des instances de CI (vérification de typage additionnel, cardinalité secondaire, ...). Chaque valeur d'une propriété d'un élément M1 (une instance de classe) est stockée dans un `FlagType`. Cette classe permet d'associer une cardinalité (secondaire) à chaque valeur. Les propriétés des éléments M0 sont gérées par les classe `FlagSet` et `Flag` (détaillées en 5.6.2).

5.4 Bootstrap



La création du noyau de RAM3 est effectuée, au démarrage, par les classes contenues dans le paquetage `init`. Une implémentation statique du noyau, par rapport à un stockage sur fichier, a pour inconvénient d'être moins évolutif : un fichier ne nécessite pas de compilation, et demeure en général plus facile à lire et à comprendre qu'un ensemble de classes Java. Néanmoins, l'utilisation d'un *bootstrap* programmé est privilégiée pour deux raisons :

- Le méta-méta-modèle MOF se définit lui-même. Certaines définitions sont fortement réflexives. Par exemple, une classe dispose d'un ensemble de propriétés qui sont en fait des instances d'attributs et de références, eux-mêmes des instances respectives des classes Attribut et Référence qui sont elles-mêmes des classes. De telles définitions auraient été difficiles à écrire dans un fichier (même en utilisant des pré-définitions partielles, comme en C ou en IDL). L'utilisation des fonctions de *bootstrap* ont permis de construire au petit à petit le méta-méta-modèle MOF et de factoriser un grand nombre de propriétés (dont les valeurs étaient les mêmes pour beaucoup d'éléments).
- Les variables de classes Java qui ont servi à définir le (méta-méta-) modèle MOF peuvent être très utiles pour accéder aux éléments de ce modèle à partir du langage Java. Par exemple, *ram3.init.MOF.Model.Namespace.contents* correspond à la référence *contents* de la méta-entité *Namespace*[78].

[78] L'invocation `get("MOF.Model.contents.Namespace.contents.contents")` correspondante est plus longue en temps d'exécution et peut générer une exception Ram3Exception. La gestion d'une exception nécessite la présence des instruction try et catch, ce qui alourdit l'implémentation.

5.5 Le comportement

5.5.1 Le fonctionnement

Que ce soit pour les opérations internes, les méthodes d'affichage ou l'accès Corba, les méthodes d'un Ram3Object qui sont en rapport avec son comportement (dans RAM3) ne sont qu'une redirection vers des objets traitements. L'exemple suivant illustre en détail ce mécanisme :

Core.Space est le type (et le méta-objet) des éléments *Root*, *Core* et *MOF* (contenant *Model*, *Reflective* et *Facility*) de RAM3. L'invocation de `add(Ram3Object newElement)` sur *Root* correspond 1) à l'ajout du *newElement* dans la liste `contents` et 2) à l'affectation de *Root* au `container` de newElement. Ce traitement est celui défini par la classe `ram3.inner.Core.Space`. *Core.Space* référence une instance de cette classe à travers l'attribut `inners`. Lorsque `add` est invoqué sur *Root* (cf. figure 87), c'est la méthode `set` de l'objet traitement associé au méta-objet de *Root* qui est invoquée. Le méta-objet de *Root* étant *Core.Space*, c'est la méthode `add` de l'instance de la précédente classe `ram3.inner.Core.Space` qui est invoquée. Pour effectuer cette invocation, il est demandé à la précédente classe, pour des raisons de typage Java, d'implémenter l'interface (`ram3.inner.interfaces.`)`AddInterface` qui définit la méthode `add`.

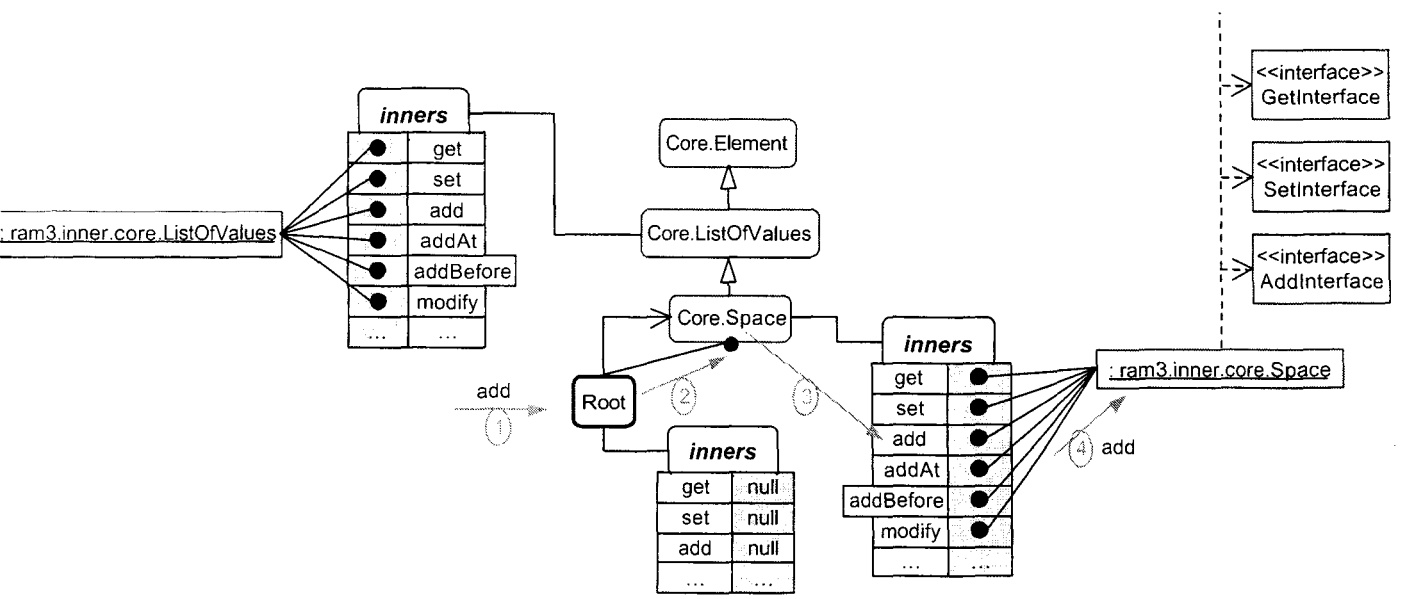


figure 87. Gestion du comportement de Root

5.5.2 Les opérations internes

La liste des opérations internes est principalement constituée des opérations définies par la méta-entité *Reflective.RefObjet* : l'instanciation, la lecture/modification/suppression d'une propriété et l'invocation (`add`, `addAt`, `addBefore`, `create`, `get`, `invoke`, `modify`, `modifyAt`, `remove`, `removeAt`, `set`). Nous avons, par exemple, ajouté à cette liste une méthode d'affichage `ToString` déjà présente dans la classe `Object` du langage Java. Il s'agit de l'affichage textuel d'un objet. Le tableau 2 montre le reste des opérations internes.

DefaultValue	La valeur par défaut pour les instances de l'objet. Cette opération sert pour les attributs, les références et les types de bases.
GiveContents	Donne le contenu d'un élément. La constitution du contenu est différent selon le type de l'objet (suivant les liens de composition).
IsA	Teste si un élément est instance d'un autre.
Move	Permet de changer la place un élément d'une liste.
RealQualifiedName	Nom qualifié réel. Ex : *Dare.contents.Task.contents.uses*
ToString	Affichage textuel d'un élément

tableau 2. Autres opérations internes

La classe de traitement la plus conséquente est celle associée à la méta-entité *Class*. Elle gère le comportement des classes et des instances de celles-ci[79]. La figure 88 montre l'algorithme simplifié de l'opération interne set de la méta-entité Class. L'intérêt de cet algorithme est de montrer le nombre de tests à effectuer pour une simple affectation. La représentation de l'algorithme n'indique pas les tests sur l'interdiction d'affectation (*isChangeable* ou *isFrozen*) ou sur le fait que la propriété trouvée est peut-être une propriété dérivée (*isDerived*). Il n'indique pas non plus la gestion des cardinalités secondaires. Cette complexité provient essentiellement du grand nombre de propriétés des méta-entités.

Le comportement des éléments M-zéro, défini par l'entité *ClassInstanceInstance*, hérite de celui de Class. Elle ajoute dans chacune des opérations les mécanismes additionnels propres au niveau M-zéro.

[79] La classe *ClassInstance*, que l'on a pu voir précédemment, hérite de *Class* et ne modifie pas le comportement défini par *Class*.

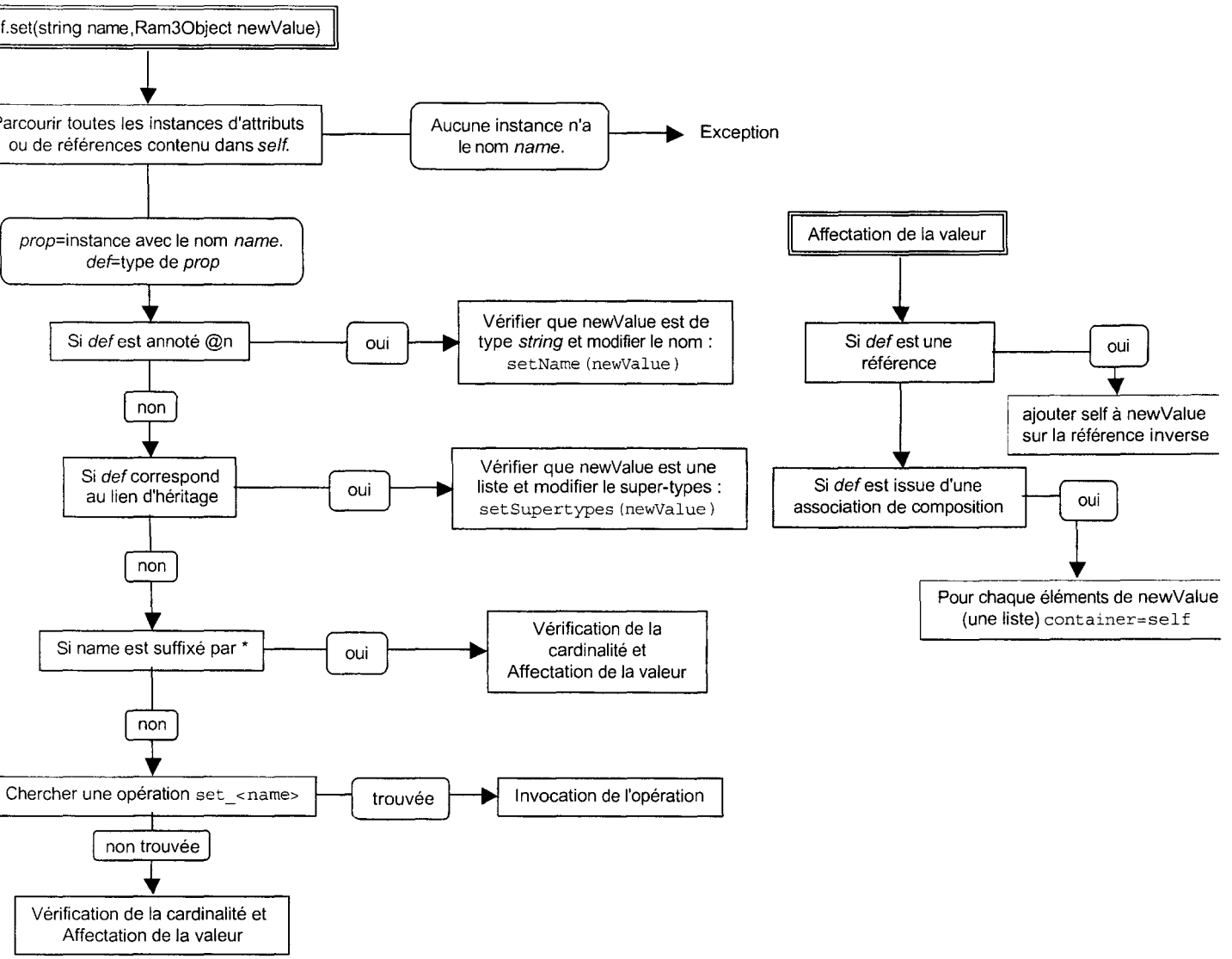


figure 88. "Algorithme" de l'opération interne *set* de *Class*.

5.5.3 L'affichage

Quatre méthodes d'affichages, que nous avons déjà évoquées, existent. Chacune d'elles est illustrée sur la figure 89. Si pour les opérations internes c'est la classe traitement du méta-objet qui est utilisée, pour les méthodes d'affichage c'est la classe traitement du type. Lorsque RAM3 affiche la liste des classes de *Dare*, il invoque pour chacune d'elles la méthode `shortDisplay` (définie donc dans *Class*). Quand il affiche la classe *Task*, c'est la méthode `longDisplay`. Pour l'ajout d'une classe, la méthode `graphicalInstanciation` de *Class* est invoquée afin de créer une classe. Enfin la modification "graphique" d'une valeur non-objet est gérée par la méthode `edit` de son type.

ex. M2

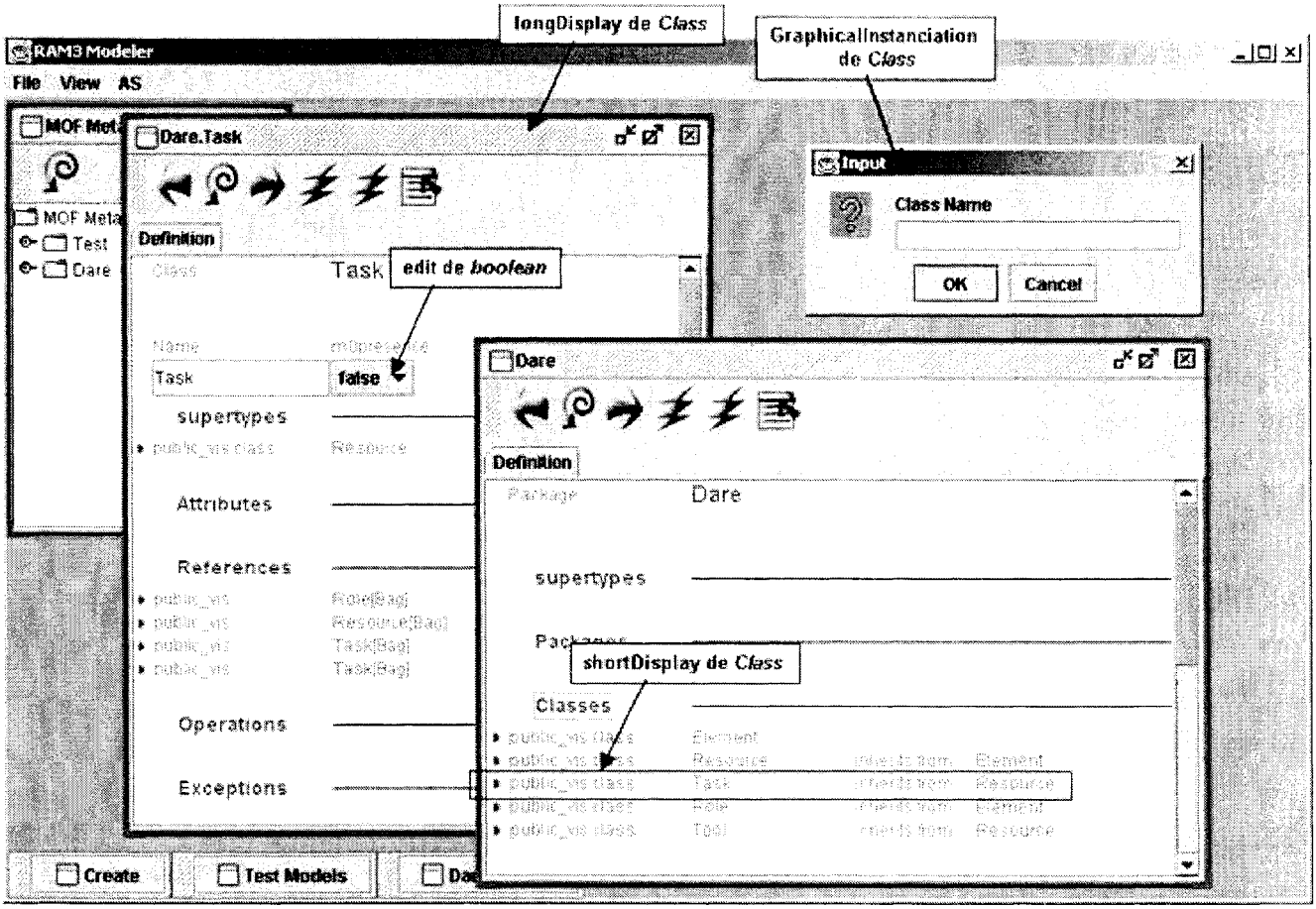
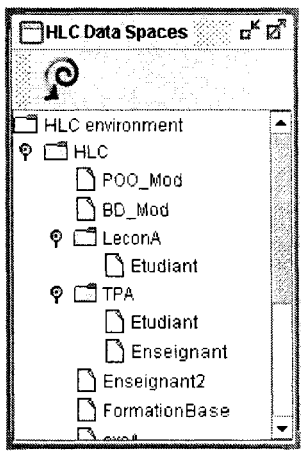


figure 89. Les différentes méthodes d'affichage

Les explorateurs permettent un accès rapide à un élément. Chaque item correspondant à un élément. L'item peut être déroulé s'il contient des éléments. Son contenu est donné par l'opération interne *giveContents*. Pour une instance de classe, comme *TPA*, le contenu est égal à l'ensemble des éléments référencés par les liens de composition. Sur l'image à droite, *TPA* contient *Etudiant* et *Enseignant* car l'association *TaskRole* est ici définie comme composite (ce qui n'est pas le cas en réalité). Pour une classe, comme *Task*, seul le lien *contents* est de nature composite.

ex. M1

ex. M2



5.5.4 L'accès Corba

Chaque élément de RAM3 est accessible à distance, à travers un bus Corba. Les interfaces IDL des éléments suivent la projection MOF vers IDL vu dans le chapitre 5. Ainsi il est possible "d'invoquer" (de l'extérieur de RAM3) l'opération IDL `add_uses` définie par l'interface IDL `Task` sur l'objet *TPA*. Deux fonctionnalités de RAM3 permettent cette interface Corba :

ex. M1

1. La génération des interfaces IDL pour un paquetage MOF
2. La connexion des éléments de RAM3 a un bus Corba et l'utilisation du DSI (*Dynamic Skeleton Invocation*) qui permet de traiter directement une requête/invocation Corba.

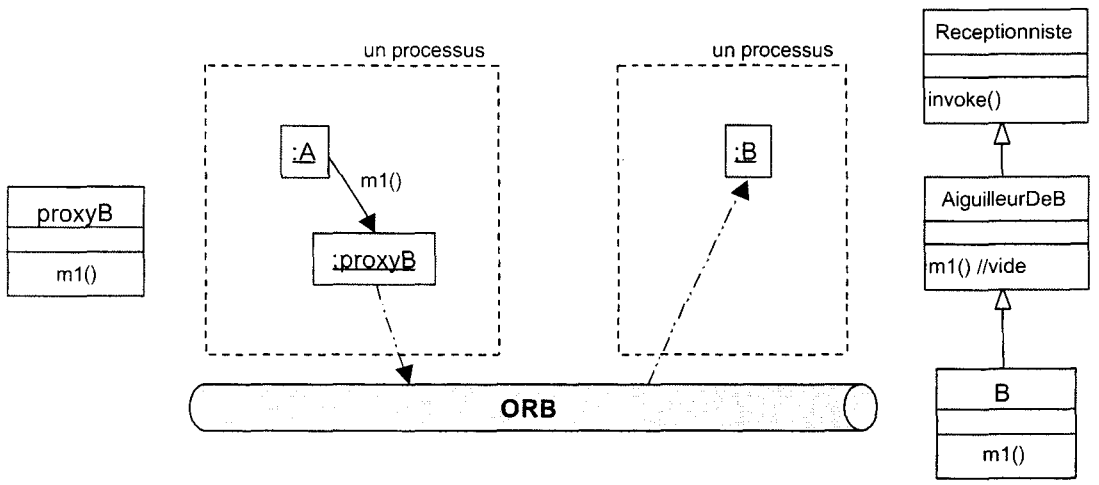


figure 90. Principe des invocations Corba

La génération des interfaces est une simple implémentation des règles de projection MOF vers IDL. L'implémentation est répartie sur les différentes classes de traitement Corba. En effet, chaque Ram3Object dispose d'une méthode `giveIdlDefinition`. Le traitement associé à cette méthode est spécifique à chaque type d'objet et est défini dans une classe Java particulière. Par exemple, la classe `ram3.corba.mof.model.Class` fournit, pour une classe MOF donnée, la génération des deux interfaces IDL correspondantes à la classe (*Fabrique* et *Instance*).

La connexion avec un bus Corba et la gestion des invocations est plus délicate. Pour expliquer celle-ci, il faut rappeler brièvement le fonctionnement d'un bus Corba. Un bus Corba permet d'invoquer un objet distant (situé sur une autre machine ou simplement dans un autre processus mémoire) comme s'il s'agissait un objet local, quels que soient les langages utilisés pour effectuer l'invocation et celui de l'objet distant. La structure employée pour gérer cette hétérogénéité d'environnements n'est pas toujours la même selon le bus Corba utilisé. Mais le principe reste le même. Nous avons utilisé l'ORB Orbacus (3.0) [ION 02] dans sa version Java. La figure 90 illustre, sur l'exemple de l'invocation de la méthode *m1* d'une instance de *B* par une instance de *A*, la structure adoptée par cet ORB.

Principe de fonctionnement des invocations Corba sous Orbacus : l'accès à une méthode distante est géré grâce à une classe proxy (du côté client) qui transmet l'invocation à l'objet distant (côté serveur) et grâce à une classe de désemballage qui traduit chaque invocation distante reçue en une invocation sur le réel objet.

Pour que la classe B soit accessible à distance, le concepteur définit auparavant la liste des opérations distantes dans une interface IDL. Ici, elle contient l'opération IDL `m1`. Le compilateur IDL d'Orbacus/Java traduit cette interface vers le langage Java. Plusieurs classes Java sont générées. Les deux principales sont `_BImplBase` (notée *AiguilleurDeB* sur la figure) et `BHelper` (notée *proxyB*). Lorsque *a*, l'instance de *A*, invoque "virtuellement" la méthode *m1* sur *b*, un objet de type *B*, il invoque en fait la méthode *m1* sur une instance de *proxyB*. Cette dernière envoie sur le bus Corba une requête "m1" destiné au réel objet *b*. Grâce aux informations, contenues dans la requête, sur l'emplacement physique et logique de l'objet b, le bus Corba envoie la précédente requête à *b*. Dans la perspective d'instances accessibles à distance, le concepteur a dû faire hériter la classe *B* de la classe *AiguilleurDeB*. Celle-ci hérite elle-même de *Receptionniste* (`DynamicImplementation` en réalité). *Receptionniste* définit (en autres) la méthode `invoke`, invoquée par le bus Corba. La méthode `invoke` a comme paramètre le nom de la méthode invoquée et les arguments. Cette méthode est redéfinie dans *AiguilleurDeB* pour invoquer la méthode `m1` et renvoyer, au bus, le résultat de celle-ci (déjà définie dans *AiguilleurDeB*) si le nom de la méthode passé en argument est "m1". Comme la classe *B* redéfinit `m1`, c'est la méthode `B.m1` qui renvoie le résultat. Les classes *proxyB* et *AiguilleurDeB* effectuent le typique emballage/déballage de requêtes [MIC 99]. Ces classes sont

respectivement appelées la souche et le squelette. La structure d'Orbacus utilise le DSI, c'est-à-dire la classe `DynamicImplementation`.

Utilisation du DSI d'Orbacus dans RAM3. RAM3 redéfinit le désemballage (classes *Aiguilleur*) pour traduire une invocation Corba en invocation RAM3/MOF.

Pour un paquetage (i.e. un méta-modèle) MOF, RAM3 génère les interfaces IDL correspondantes. Seules les souches créées (i.e. la partie cliente) par leur compilation IDL sont utilisées. Par contre, l'aiguillage de la requête (i.e. la partie serveur) est effectué directement par les opérations d'accès Corba spécifiques à chaque objet (cf. figure 91 partie basse). Chaque Ram3Object dispose d'une instance de la classe `CorbaProxy` qui hérite de `DynamicImplementation`. La méthode `invoke` de ce `CorbaProxy` est redéfinie afin d'invoquer la méthode `invoke_CORBA_` du Ram3Objet auquel ce proxy est associé. Cette méthode invoque l'opération `invoke_Corba` de l'objet traitement "Corba" associé au méta-objet du Ram3Objet. Par exemple, la méthode `invoke_Corba` de la classe de traitement Corba pour les classes MOF (`ram3.corba.mof.model.Class`) traduit les requêtes Corba dont le nom de la méthode est `set_<name>` en invocation `set(<name>,...)` sur le Ram3Objet (cf. figure 91 partie haute).

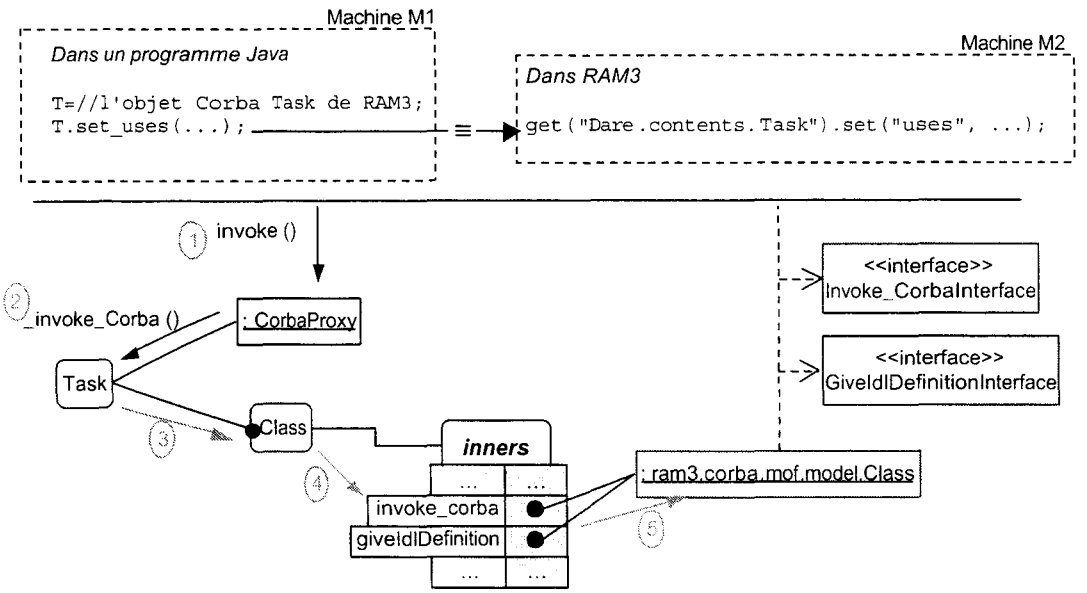


figure 91. Fonctionnement de la réception d'une requête Corba

5.6 L'intercession

5.6.1 Objectif

Dans la partie 2, nous indiquons que RAM3 dispose de mécanismes d'intercession, c'est-à-dire la capacité de changer les types d'éléments déjà existants et les opérations internes. Ces mécanismes sont généralement assez lourds à implémenter, ils le sont encore plus avec les nombreuses propriétés des classes ou de la définition de leurs caractéristiques. Par exemple, un attribut est caractérisé par sept propriétés : le nom, le type, la cardinalité, la dérivabilité, la possibilité de modifier la valeur, la visibilité et la portée (attribut d'instance ou de classe). Quatre autres propriétés sont ajoutées par les extensions M0. Il y a donc 11 types de modifications possibles qui impliquent 11 types de répercussion sur les instances de la classe (contenant l'attribut).

Les répercussions se font sur deux niveaux : les instances de classes et les instances de ces instances (éléments M0). La suppression de la référence *involves* de *Task* supprime *Enseignant* et *Etudiant* de *TPA* et supprime aussi *EnseignantTPA01_02* et *EtudiantTPA01_02* de *TPA01_02*. Cette double portée des modifications augmente la complexité des mécanismes d'intercession.

ex. M2→M1,M0

Vis-à-vis de l'intercession, nous aurions pu ajouter des fonctions d'évaluation des modifications, ajouter la possibilité d'associer une règle de transformation des valeurs lorsqu'on modifie le type d'un attribut, mettre en place des verrous ou un système de condition pour limiter les modifications. Mais notre objectif était de fournir les mécanismes d'intercession élémentaires pour le prototypage de méta-modèles. La perspective d'utiliser de tels mécanismes pour la maintenance d'un méta-modèle (i.e. avec un grand nombre d'instances) aurait par contre nécessité une étude plus approfondie sur des fonctionnalités plus avancées.

5.6.2 Principe de répercussion

Les problèmes de répercussion sont principalement associés aux caractéristiques structurelles (attributs et références). Lorsqu'un attribut ou une référence est modifié, toutes ses instances (de type *AttributeInstance* ou *Link*) sont modifiées en conséquence. Comme ces instances sont modifiées, leurs propres instances (de type *Flag*) sont elles-aussi modifiées. La figure 92 montre l'architecture qui gère la référence *involves* sur les 3 niveaux : comme une définition de référence dans la classe *Task*, comme une référence de la tâche *TPA* et comme une propriété de l'instance de *TPA:Activity[TPA01_02]*.

ex. M2 ex. M1 ex. M0

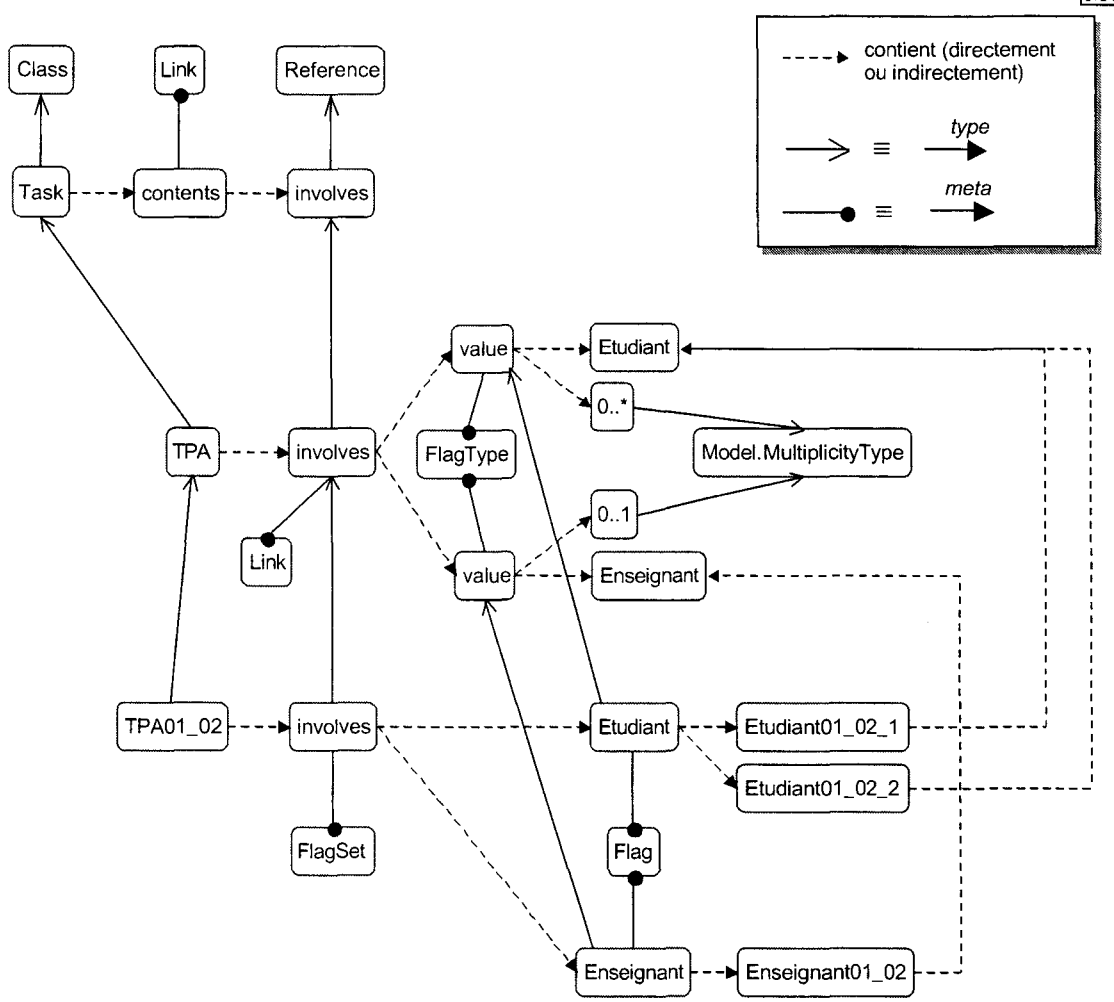


figure 92. La référence *involves* à travers les trois niveaux

ex. M2

La référence *involves* issue de l'association *TaskRole* fait partie de la définition de la classe *Task* : elle est référencée à travers la propriété *contents* de *Task*. *Involves* devient, de ce fait, une propriété pour les instances de *Task*. C'est la création d'une instance de la référence *involves* (qui donne un Ram3Object dont le méta-objet est *Link*) qui réalise cette propriété.

ex. M1

TPA contient (via l'attribut `container`) un tel objet pour réaliser sa propriété *involves*. Chaque type de rôle *impliqué* par *TPA* est référencé par le précédent objet, mais aussi à laide d'un Ram3Object intermédiaire : un *FlagType*. Cet objet permet d'associer une cardinalité (secondaire) à chaque valeur.

ex. M0

La structure des propriétés des éléments M0 est légèrement plus complexe. Pour *TPA01_02*, la propriété *involves* est réalisée par une instance du lien *involves* de *TPA*. Les instances de liens ou d'*attributeInstance* ont pour méta-objet *FlagSet*. Un *FlagSet* contient une liste de *Flags*. Chaque *flag* gère un point d'entrée secondaire. Par exemple, pour *TPA01_02.involves*, il y a un *flag* pour la liste des étudiants et un *flag* pour l'enseignant. Chaque *flag* est une instance du type secondaire correspondant. Par exemple, le *flag* contenant la liste des étudiants est une instance du *flagType* contenant le rôle *Etudiant*.

La répercussion d'une modification suit toujours les liens d'instanciation :

ex. M1→M0

- La suppression du type de rôle *Etudiant* de *TPA* aura pour répercussion d'effacer toutes les instances du *FlagType* associé à *Etudiant*.
- L'ajout d'un rôle *Assistant* à *TPA* crée un *flag* dans chaque *FlagSet* instance du lien *involves* de *TPA*. Chaque *flag* créé est une instance du *FlagType* associé à *Assistant*.
- La modification de la cardinalité secondaire (de 0..* à 0..1) associée à *Etudiant*, "supprime" une par une les valeurs contenues dans chaque instance du *flagType* attaché à *Etudiant*. Un seul étudiant sera, par la suite, accepté.

ex. M2→M1,M0

Comme nous l'avons dit précédemment, la modification d'un attribut ou d'une référence, comme *involves*, a une double répercussion. Supprimer la référence *involves* entraîne la suppression de tous les liens *involves*. Pour chaque lien supprimé, sont aussi effacées ses instances (*flagSet*). Quand le type de la référence *involves* (ex : *Resource*[80] à la place de *Role*) est changé, tous les *flagTypes* contenus dans les liens sont effacés. Les instances de chaque *flagType* (*flag*) sont aussi effacées.

La majeure partie des propriétés d'un attribut (*multiplicity*, *isDerived*, *isChangeable*, *m0presence*, *type*, *name*, *attributeForMultiplicity*) ont un mécanisme de répercussion associé. Pour les références, qui dispose sensiblement des mêmes propriétés (dont une partie est définie dans les *extrémités/AssociationEnds* de l'association), la situation est identique.

ex. M2→M1,M0

La modification d'une association implique une autre opération : si le type d'une extrémité de l'association est changé, l'ancien type se voit supprimer sa référence. Si par exemple, l'association *TaskRole* relie maintenant *Task* et *Resource* (!) la référence *realises*[81] est supprimée de *Role* et ajoutée à *Resource*. La suppression et l'ajout impliquent les mécanismes précédents de répercussion (*Reference→Link→FlagSet*).

La suppression ou l'ajout de lien d'héritage utilise aussi les mécanismes de répercussion associés aux caractéristiques. Dans le nouveau méta-modèle DARE, *Task* hérite de *Resource*. Supprimer ce lien d'héritage implique que toutes les caractéristiques définies par *Resource* (et les super-types de celui-ci) n'apparaissent plus dans les instances de *Task* (et dans les activités). Si *Task* re-hérite de *Resource*, les propriétés ré-apparaîtront (sans leur contenu). ex. M2

6 Réalisation pratique de RAM3

Le thème de recherche l'équipe NOCE est le travail coopératif assisté par ordinateur (TCAO) avec comme spécialité l'enseignement à distance (EAD) (cf. chapitre 1). Les travaux s'orientent donc sur les interactions homme machine, la psychologie et l'étude des sciences sociales. Les réalisations sont des systèmes pour supporter les activités coopératives, des ateliers de conception d'interfaces graphiques ou sonores, de synthèse vocale, ... RAM3 est très général et ne porte pas sur le précédent

[80] De la nouvelle version du méta-modèle DARE.

[81] La référence de *Role* qui est l'inverse de la référence *involves* de *Task*.

champ applicatif. L'implémentation de RAM3 a donc été réalisé par l'auteur du manuscrit seul. Il en est de même pour l'étude du MOF (domaine nouveau pour l'équipe[82]).

Le développement de RAM3 a demandé un gros travail d'implémentation. La première version contenait 36000 lignes de code (commentaires inclus). La deuxième version (re-développé à partir de zéro) dont le but était d'intégrer les extensions M-zéro n'en contient "que" 31000 lignes. Pourtant elle intègre beaucoup plus de fonctionnalités (personnalisation des interfaces plus simples, shell plus puissant, ...). Un gros effort de factorisation a été effectué afin de rendre le code de RAM3 plus compact.

En plus d'une connaissance approfondie du MOF, le développement a necéssité une réelle expertise :

- du langage Java : les mécanismes d'introspection, utilisation du compilateur à partir d'un programme Java, l'API Swing, ...
- du support Corba : les principales difficultés viennent de l'absence de documentation de l'utilisation du DSI (sur un ORB précis).
- du langage XML : syntaxe du langage et utilisation des API DOM et SAX pour parser et écrire des documents XML.

Enfin différents stages ont eu lieu autour de RAM3. Nous avons évoqué un stage sur le support WSDL/SOAP. Il y en a deux sur le support XMI et un sur les interfaces graphiques (qui n'a malheureusement pas abouti).

7 M-CAST

7.1 Un développement en cours

RAM3 est un outil MOF qui répond à nos attentes vis-à-vis de M-CAST et du prototypage de méta-modèles. Il propose :

- Éditeur-navigateur de modèles et inspecteur d'objets MOF.
- Plusieurs méta-modèles chargés simultanément.
- Modulaire et ouvert : se servir des fonctionnalités de l'outil à partir d'un langage de programmation, et possibilité d'intégrer de nouveaux modules, de nouvelles fonctionnalités au sein de l'outil (comme M-CAST).
- Un réel éditeur MOF (méta-modèles et modèles) avec des caractéristiques réflexives.
- L'intégration du niveau M0 dans l'éditeur MOF "réflexif" constitue un apport considérable.

Il nous reste maintenant à étudier l'implémentation de M-CAST à l'aide de RAM3. Au moment où nous rédigeons ce manuscrit, le développement de M-CAST n'est pas encore terminé. Une des raisons est l'implémentation conséquente de RAM3. Les extensions pour M0 ont demandé aussi du temps pour leur définition et leur implémentation (et bien sûr pour les prototyper). De plus, pour l'ensemble de nos travaux en rapport avec CAST, nous avons préféré nous focaliser sur l'approche plutôt que le développement en lui-même. Ce choix est d'abord une conséquence de la situation actuelle : il y a encore peu de systèmes de groupware flexibles (méta-modèle ou non) en réelle utilisation industrielle et l'utilisation du MOF dans de tels systèmes (et même ailleurs) est encore inexistante. Dans ce contexte, la validation de notre solution est difficile. Pour cette raison, l'implémentation de M-CAST et la phase de tests ont été secondaires. Néanmoins, l'implémentation est bien entamée et devrait être finie dans un avenir très proche (cf. chapitre 7). Nous présentons ici la

[82] Le rapprochement de l'architecture de DARE et de la structuration à trois niveaux n'a été faite qu'après l'étude du MOF.

partie la plus avancée : l'éditeur de liens d'adaptation. Nous ne parlerons pas de l'implémentation du paquetage CAST qui ne fait "que" suivre le modèle UML présenté dans le chapitre 3.

7.2 L'éditeur de liaisons graphiques

L'éditeur de M-CAST permet d'effectuer la majeure partie des liens d'adaptations :

- liens entre entités avec une éventuelle condition (exprimée en Java),
- liens entre caractéristiques avec la saisie éventuelle de fonctions d'adaptation (aussi exprimée en Java) pour les différents types d'accès.

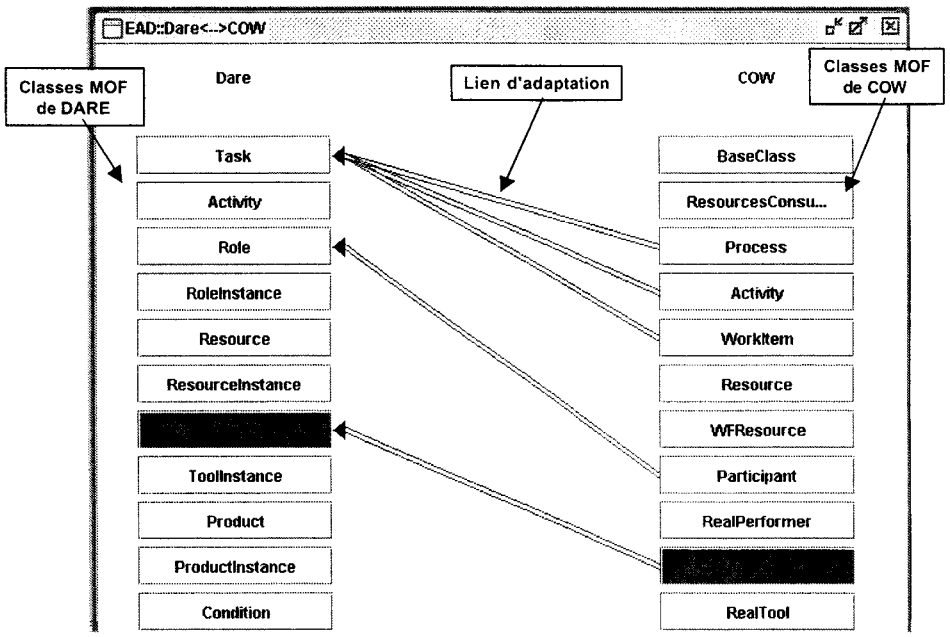


figure 93. Éditeur de liaisons d'entités

La création de fonctions de transformations pour les types de bases n'est pas encore gérée ainsi que la pose de verrous sur les types d'accès d'une caractéristique.

Les figure 93 et figure 94 illustrent l'utilisation de l'éditeur. Après avoir choisi deux méta-modèles, l'éditeur affiche une fenêtre pour tisser les liens entre les classes MOF. Cette fenêtre affiche sur deux colonnes l'ensemble des classes MOF de chaque méta-modèle (cf. figure 93). Après avoir sélectionné deux classes de méta-modèles différents, il est possible de créer un lien d'adaptation et d'adjoindre à celui-ci une condition. L'éditeur peut afficher, dans une fenêtre, le détail des liens entre caractéristiques d'une liaison entre deux classes MOF (cf. figure 94). Cette nouvelle fenêtre dispose d'un sens qui est celui de la liaison entre les deux classes. Les liens entre caractéristiques suivent toujours ce sens. Les liens sont regroupés par la caractéristique cible. Des fonctions d'adaptations pour chaque type d'accès peuvent être définies pour chaque (groupe de) lien(s).

ex. M2

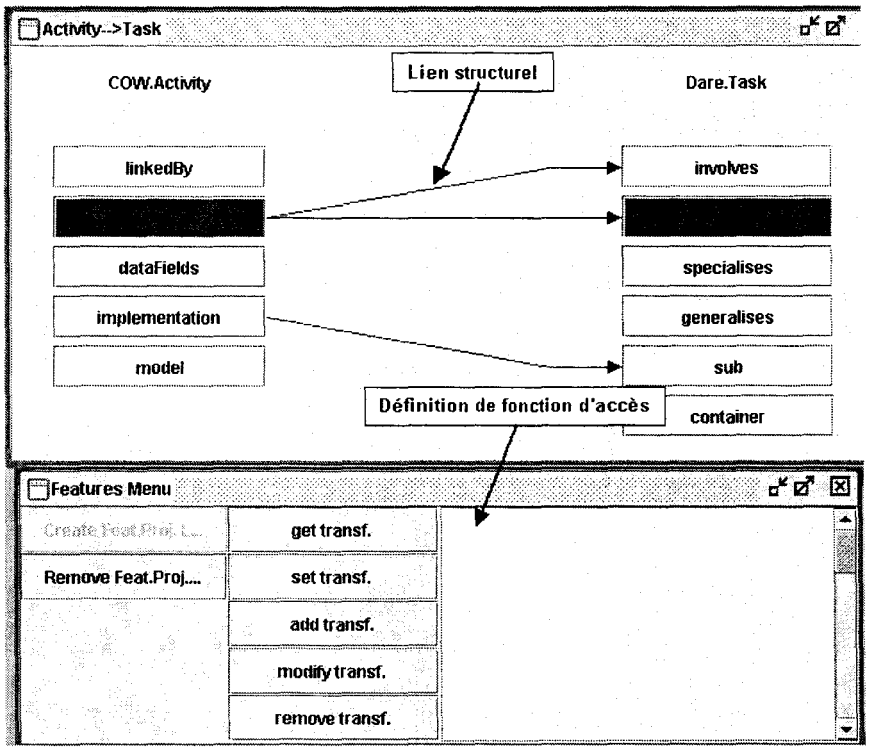


figure 94. Éditeur de liens structurels

L'éditeur profite bien sûr pleinement de l'API de RAM3 pour charger des méta-modèles en mémoire et pour les manipuler grâce aux opérations internes des *Ram3Object.*

7.3 Implémentation en cours

Sur les quatre parties de M-CAST, décrite en 1.1, les deux dernières restent à développer : la génération des interfaces IDL pour les paquetages *X_Adapter*, et la génération/Association du code d'implémentation (Java). Seule l'implémentation du paquetage CAST (cœur de tout service d'adaptation) est bien avancé. La génération de l'implémentation des paquetages *X_Adapter* est à l'état d'embryon tout comme la génération des interfaces IDL. Le principe de cette dernière ne présage pas de difficultés majeures. Pour le prototypage de services d'adaptation dans RAM3, les méta-modèles *X_Adapter* contiendront des classes dont les opérations d'accès seront le code de redirection contenu dans l'implémentation finale générée par M-CAST.

8 Conclusion

Après avoir exprimé les besoins en outillage de notre application de CAST sur le support MOF, nous avons conclu que les outils MOF actuels n'étaient pas suffisants pour ces besoins. Des perspectives d'un prototypage de méta-modèles plus facile, et d'une assistance plus grande aux néophytes de la méta-modélisation MOF nous ont définitivement décidé à proposer un nouvel outil MOF dont l'objectif est la dynamicité. L'outil développé à cette fin, RAM3, répond aux nécessités de modularité, de réflexivité, et d'interfaces d'édition rapide de méta-objets MOF. Il intègre de surcroît les extensions M0 définies dans le chapitre 5. Cela nous permet de proposer une implémentation de ces extensions, de faire des tests de celles-ci et de disposer pour M-CAST d'un outil MOF qui gère les trois niveaux. La taille conséquente du développement de RAM3 est à l'origine du non-achèvement de l'implémentation complète de M-CAST. Néanmoins, nous avions insisté sur le fait que l'approche conceptuelle était privilégiée et que le fait qu'il n'y ait pas encore de systèmes utilisant le MOF empêche de réels tests de mise à l'épreuve de services d'adaptations issus de M-CAST.

Chapitre 7

Bilan des travaux & Perspectives

1 Bilan

1.1 Les objectifs de départ

La coopération entre systèmes informatiques est un enjeu industriel capital. Les groupware, et principalement les WfMS, ont connu une forte expansion dans le début des années 90. Des travaux sur la coopération de tels systèmes, comme ceux du WfMC, sont la preuve de l'importance de la coopération de groupware et de celle de la coopération inter-organisationnelle en général. Néanmoins, ils ne prennent pas en compte le caractère flexible de la nouvelle génération de groupware. Les solutions d'interopérabilité (moins spécifiques que les solutions de coopération), qui sont plus nombreuses, ne prennent pas non plus en compte la flexibilité (cf. chapitre 2), c'est-à-dire la capacité pour les non-informaticiens de modifier dynamiquement le modèle d'exécution du système. Aucune d'entre elles ne respecte les quatre contraintes que nous avions identifiées (cf. chapitre 1) dans le cadre de la coopération de groupware flexibles : accessibilité, visibilité et flexibilité des liens de coopération et un champ d'action plus faible.

Nos travaux ont donc pour objectif d'apporter une solution à ce nouveau type d'interopérabilité. Plusieurs "approches" existent pour construire un groupware flexible. Nous avons choisi de nous focaliser sur la plus puissante potentiellement : l'approche par méta-modèle. Cette approche naissante semble la plus prometteuse. La méta-modélisation permet de définir un modèle de fonctionnement à partir de concepts spécifiques à un domaine. Les concepts peuvent aussi être adaptés aux utilisateurs d'un système particulier. La méta-modélisation implique dans le système l'existence d'une structure à trois niveaux : le méta-modèle, les modèles et les instances de modèles.

À partir de ce choix, nos objectifs sont fixés :

- Fournir une solution pour la coopération de groupware flexibles qui répondent aux quatre contraintes précitées.
- La solution doit être adaptée aux structures à trois niveaux. Elle doit tirer profit des liens forts existants entre les modèles et leurs instances.

1.2 Satisfaction des enjeux

1.2.1 Une approche conceptuelle

Dès le départ, nous savions qu'il eut été difficile de valider toute proposition. La méta-modélisation s'apparente plus à un paradigme qu'à une réelle technologie : les groupwares adoptant l'approche par méta-modèle n'ont en commun que la structure à trois niveaux. Les langages ou les

environnement utilisés sont différents. Dans notre démarche, il s'agissait plus de poser les bases de la coopération entre de tels systèmes. La phase de validation, difficilement réalisable actuellement, a été mise de côté. Par conséquent la principale partie de notre proposition est d'abord un cadre *conceptuel*, CAST, qui n'est attaché à aucune plate-forme particulière.

CAST a pour but d'adapter les modèles et leurs instances d'un système coopérant à l'autre. De cette manière, ces modèles et instances apparaissent dans un environnement différent du leur, et peuvent ainsi être associés avec des éléments extérieurs. Pour que ces associations externes soient visibles dans l'environnement d'origine des précédents modèles, il faut que l'adaptation se fasse dans l'autre sens : les éléments extérieurs doivent aussi être adaptés dans l'environnement d'origine. Pour deux systèmes coopérants, l'adaptation est réalisée grâce à un service d'adaptation spécifique aux méta-modèles de chaque système. Le cadre conceptuel décrit les étapes de construction des services d'adaptation.

Cette adaptation (bi-directionnelle) permet de lier les systèmes à l'aide de leur propre outil de modélisation. Les liaisons avec l'extérieur sont ainsi :

- accessibles pour les utilisateurs. Grâce à l'outil de modélisation, avec lequel ils sont familiers, ils peuvent effectuer des liens avec des modèles externes.
- visibles. Les modèles sont certes liés avec des adaptateurs d'éléments externes mais ils demeurent liés avec ces éléments. Ces liens apparaissent dans l'outil de modélisation. Ils sont donc visibles aux utilisateurs qui en ont conscience lors qu'ils effectuent des modifications.
- flexibles. Effectués par les outils de modélisation, les liens de coopération ne nécessitent pas l'intervention des développeurs et peuvent être réalisés dynamiquement.
- Effectués à partir d'un champ d'action plus faible. Les liens de coopération ne requièrent pas d'accéder à l'implémentation du système.

À travers les notions de concepts de type et d'instance, nous avons décrit les liens, les plus fréquents, existants entre les modèles et leurs instances. CAST intègre ces relations. Un grand nombre de liens d'adaptation pour les instances de modèles sont ainsi déduits des liens entre les deux méta-modèles et les modèles. De ce fait, nous tirons profit de la spécificité de l'architecture à trois niveaux.

Conceptuellement, nous avons satisfait nos deux enjeux.

1.2.2 Implémentation et outillage MOF

Même s'il est difficile de valider CAST à cause de la diversité (possible) des supports de méta-modèles, nous avons toutefois commencé l'application de CAST sur un support particulier : le Meta-Object Facility. Le MOF est l'initiative la plus importante pour standardiser l'utilisation de méta-modèles au sein de systèmes distribués. Dans l'application de notre "pattern" nous avons dû définir des extensions au MOF pour qu'ils prennent en compte les concepts de type et d'instance. Cette définition était nécessaire car les méta-modèles MOF n'intègrent pas la description du niveau M-zéro. Avec ces extensions, CAST peut profiter de la spécificité de l'architecture à trois niveaux en MOF. La définition des extensions M-zéro a impliqué l'extension du méta-méta-modèle MOF et des règles de projection MOF vers IDL.

Il est bien évident que l'implémentation de ces extensions M-zéro est à notre charge. Au lieu de partir d'un des rares outils MOF existants, nous avons toutefois souhaité créer un nouvel outil MOF pour plusieurs raisons. L'utilisation d'un service d'adaptation dans un contexte industriel implique une phase de tests et d'évaluation. Cette phase requiert un outil capable d'éditer facilement des modèles MOF afin de pouvoir les modifier et tester ainsi une multitude de cas. À travers notre expérience, nous avons remarqué que la méta-modélisation est une pratique qui, au premier abord, repousse les concepteurs. La nécessité de prototypage de méta-modèles MOF est par conséquent très importante. Elle doit aider le concepteur à se familiariser avec la méta-modélisation. Cette nécessité intervient une

seconde fois avec l'intégration du niveau M-zéro. Concevoir un méta-modèle qui dispose d'une double-instanciation sollicite une compétence intellectuelle rare. Pouvoir prototyper le niveau M-zéro est donc d'une grande utilité. Notre réponse aux besoins de prototypage (M-CAST, méta-modélisation, niveau M-zéro) est l'outil RAM3 que nous avons développé. Il permet de définir des méta-modèles, de créer des modèles et de les instancier (éléments M-zéro) à partir d'éditeurs/navigateurs. En plus de cette manipulation facile à partir d'éditeur, il permet aussi de modifier dynamiquement méta-modèles et modèles, en ayant des répercussions sur les uns ou deux niveaux inférieurs. RAM3 fournit donc un outil de prototypage de méta-modèles MOF efficace.

Le développement de RAM3 ayant été important, celui de M-CAST n'est pas encore terminé. Pour l'instant, seul l'éditeur graphique de liens d'adaptation est presque terminé.

1.3 Perspectives à court terme et valorisation (transfert) *

Deux parties sont à distinguer dans le développement qu'il reste à effectuer : RAM3 et M-CAST. Nous évaluons ici le temps nécessaire pour ces parties restantes.

Note : le volume de travail à effectuer est généralement estimé pour le concepteur de RAM3. Pour un étudiant qualifié, sans connaissance au départ du MOF et de RAM3, le volume de travail peut augmenter de 30 à 60 %.

1.3.1 M-CAST

L'éditeur graphique ne nécessite plus de grands efforts. La partie principale est terminée. Il reste à créer un module de persistance et à améliorer l'ergonomie qui demeure trop "brute"[83]. Ces aspects restent néanmoins secondaire vis-à-vis de la validation de notre approche. Le paquetage CAST est lui aussi bien avancé mais n'a pas été encore testé. La génération des interfaces IDL (*X_Adapter*) et de l'implémentation des paquetages *X_Adapter* et *X_Filter* est à l'état d'embryon. Les interfaces IDL sont, on l'a vu, très simples. Par contre, l'implémentation des paquetages est plus complexe. Enfin l'intégration de M-CAST dans RAM3, pour le prototypage des services d'adaptation, ne nous semble pas poser trop de problème : 1) les méta-modèles d'adaptation seraient "ajoutés" à RAM3 2) la mise en marche du service d'adaptation associé demanderait simplement deux modèles "source" 3) deux modèles adaptées apparaîtraient (et seraient ajoutés) dans RAM3 4) les éléments d'un modèle adapté pourraient être liés avec les éléments du modèle qui n'est pas leur source.

Nous estimons à trois mois homme de développement restant pour M-CAST.

Éditeur de liaisons (deux semaines)	Implémentation *X_Adapter* et *X_Filter* (deux semaines)
Paquetage CAST (une semaine)	Intégration dans RAM3 (deux semaines)
Interfaces IDL (une semaine)	Test (un mois)

1.3.2 RAM3

Le développement qu'il reste à effectuer pour RAM3 ne concerne pas que la validation de (M-)CAST. Pour cette validation, seul le *template* de génération d'interfaces IDL pour les concepts d'instance est à finaliser. Les fonctionnalités à implémenter sont en rapport avec RAM3 en tant qu'outil MOF et outil de prototypage.

1.3.2.1 L'outil MOF

Actuellement RAM3 n'implémente pas complètement les spécifications du MOF.

[83] Il faudrait par exemple séparer les concepts de type des concepts d'instance.

Association. Le principale manque est l'absence des associations dans l'édition de modèles et les interfaces IDL : pour une association, il n'y a pas d'entité qui gèrent l'ensemble des liens issus de l'association. Nous avons déjà défini les éléments à ajouter (cf. figure 95) :

- La méta-entité *Link* est (sera) renommée *ReferenceInstance.*
- Les *flagTypes* sont remplacés par des *LinkEnds* (extrémités de lien). Le comportement de ces derniers est identique à celui des *flagTypes.*
- Pour chaque modèle, un gestionnaire de liens est créé pour chacune des associations (*<nomAssociation>Manager*).
- Les instances de la nouvelle meta-entité *Link* représentent un lien dans le sens MOF(/UML) : un élément qui lie deux éléments. Un lien contient deux *LinkEnd* et est géré par le gestionnaire.
- Les *LinkEnd* continuent à être contenus dans les propriétés (plus précisément *ReferenceInstance*).

La raison d'être de la double liaison (*Link* et *ReferenceInstance*) dont est l'objet chaque *LinkEnd* est la rapidité : il n'y a pas de parcours ou de calcul à effectuer pour connaître la propriété d'un élément ou la liste des liens instances d'une association.

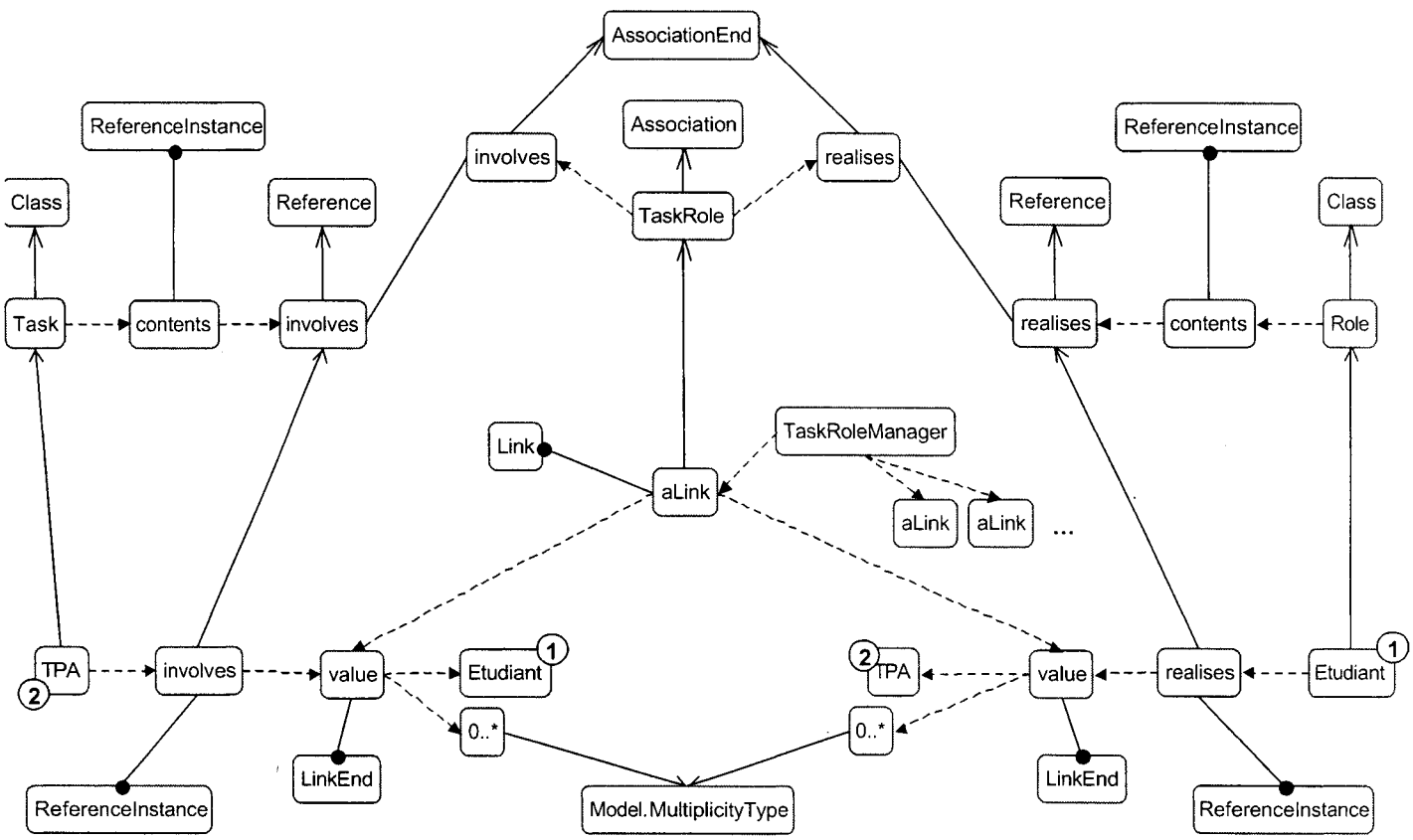


figure 95. Intégration de la gestion des liens

Le support XMI. La lecture/sauvegarde de méta-modèle au format XMI a été réalisé lors d'un stage d'étudiants en troisième cycle. Ce module utilise la DTD correspondant au MOF (les documents sont ainsi des méta-modèles). La première difficulté était de "trouver" la dite DTD. Trois DTD

différentes furent trouvées pour le MOF. Le module a été développé à partir de l'une des trois (les étudiants ayant choisi la plus facile). Ensuite nous voulions aussi lire/sauvegarder les modèles en format XMI. Actuellement cette partie, plus complexe, n'est implémentée par aucun des outils MOF existants. Elle a été commencée dans le cadre d'un second stage mené au sein du laboratoire. L'implémentation n'est effectuée qu'à 20%. Nous estimons à un mois homme le temps restant.

Pour lire/sauvegarder les éléments M-zéro en XMI, il faudra intégrer les extensions M-zéro au format XMI. À notre avis, il faut au préalable définir dans le mapping abstrait les extensions M-zéro. À partir de cette description (informelle) de l'intégration des concepts d'instance dans le MOF, l'intégration du niveau M-zéro dans XMI sera plus rapide (car guidée par des règles générales). Cette description offrira le même type d'accélération lors de l'intégration du niveau M-zéro pour les futurs supports alternatifs comme WSDL.

Le paquetage Reflective. Les mécanismes des opérations de la méta-entité *RefObject* du paquetage *Reflective* ont été implémentés pour le besoin des opérations internes. Néanmoins, les opérations héritées de *RefObject* dont dispose chaque élément MOF/Corba ne sont pas implémentées dans RAM3. Il ne reste donc qu'à rediriger les invocations Corba pour opérations réflexives vers les opérations de RAM3 correspondantes (une semaine homme).

Lecture de référentiels MOF non RAM3. L'éditeur des méta-données MOF de RAM3 ne fonctionne actuellement qu'avec des éléments de RAM3. La possibilité d'éditer des référentiels MOF/Corba non RAM3 ferait de ce dernier un outil de monitoring pour tout référentiel MOF/Corba. Pour réutiliser l'éditeur et les différentes fonctions d'affichage, nous pensons créer une classe Java qui hérite de Ram3Object et en redéfinit toutes les méthodes. Ces redéfinitions visent à traduire la requête RAM3 en requête Corba. La traduction utiliserait principalement le paquetage *Reflective*. Par exemple, l'invocation RAM3 `newInstance` donne `RefCreateInstance`, `getInstances` → `refAllObjects`, `get(<nomProp>)` → `refValue(objet de définition dans le type du nom de <nomProp>)`, ... La conversion des types (Reflective →Ram3Object) risque de présenter quelques difficultés. Pour cette raison nous estimons à un mois homme le volume de développement. L'objectif de compatibilité avec d'autres référentiels MOF en fait, à notre avis, le travail d'un partenaire industriel.

1.3.2.2 L'outil de prototypage

Pour rendre plus facile le processus de définition d'un méta-modèle et faciliter son prototypage, nous pensons améliorer l'ergonomie de l'éditeur de RAM3. Dans la liste des améliorations prévues, se trouvent des détails comme la suppression de la possibilité d'avoir plusieurs fenêtres éditant le même élément ou la modification des opérations d'affichage concernant l'instanciation pour qu'elles deviennent moins austères. Des classes MOF ne sont utiles qu'au niveau M-zéro et ne sont pourtant pas des concepts d'instance (ex : la classe *Dare.Member*). Ce type de dasses qui devraient être affichées avec les *classes_ic* apparait pourtant parmi les concepts de type. Une meilleure présentation de la structure du niveau M-zéro d'un méta-modèle fait partie des évolutions plus importantes prévues pour RAM3. Cela inclut aussi l'apparition de la structure "héritée" des concepts de type dans l'affichage des *classes_ic*.

Le formalisme de définition d'un méta-modèle MOF est soit UML soit MODL. Aucun n'est utilisé dans RAM3. Les origines de ce choix sont diverses. À la création du MOF, la représentation des "instances" (dans son sens large) dans ces deux formalismes était pauvre (pas de définition précise pour UML) ou inexistante (MODL). La représentation de modèles MOF (instances de méta-modèles) leur est donc difficile. L'apparition du niveau M-zéro (double instanciation) défavorise de surcroît leur usage. L'utilisation directe des concepts du MOF (et de leurs règles d'instanciation) dans les éditeurs de RAM3 ne pose par contre aucun problème. Ce choix est "autorisé" car les spécifications du MOF n'interdissent pas l'emploi de moyens de définition différents d'UML et MODL. Enfin cette approche nous parait coller au plus près du MOF : les deux précédents formalismes ont des bases différentes du MOF (restrictions d'utilisation pour UML, langage plus général pour MODL).

Néanmoins, la norme est aujourd'hui de décrire les méta-modèles MOF en UML. Il est vrai que la définition des liens d'héritage d'une classe ou celle des associations est plus rapide en UML que

dans RAM3, même si nous avons optimisé l'ergonomie de ces deux aspects. Il nous semble, pour cela, raisonnable d'intégrer la notation UML au sein de RAM3. Cela nous permettra d'utiliser les artefacts graphiques UML définis pour les extensions M-zéro, rendant plus aisée la définition des concepts d'instance.

Dans RAM3, la notation UML interviendrait, dans un premier temps, dans la définition de méta-modèles. Décrire des modèles et leurs instances en UML serait une seconde étape. De nouveaux artefacts seront certainement nécessaires pour bien différencier les modèles des éléments M-zéro. Le basculement -vue UML⇔vue RAM3- serait bien sûr très agréable.

Ce travail nous semble assez conséquent. Nous l'évaluons à trois mois homme pour une première version.

2 Notre apport

Nous avons évoqué/résumé la réponse que nous avons apportée au problème de la coopération de groupware flexibles. Mais quels sont les apports de nos travaux, d'une manière plus générale, et quels sont les domaines concernés ?

2.1 La dynamicité dans l'interopérabilité

L'étude des solutions d'interopérabilité nous a permis de montrer que la coopération de systèmes flexibles n'est pratiquement pas étudiée. Alors qu'il s'agit d'un véritable problème ! L'implication des utilisateurs dans la conception d'application ou de système est déjà assez récente, mais l'intervention dans la coopération de systèmes est vraiment toute neuve. Techniquement supporter une telle caractéristique ne peut pas être réalisée par les solutions d'interopérabilité existantes : il n'est pas possible de demander aux informaticiens de concevoir des systèmes basés sur le même méta-modèle et le même accès informatique (normalisation) ; l'usage de médiateurs externes ne permet pas aux utilisateurs d'avoir conscience d'une coopération et ne leur offre pas par conséquent la possibilité de définir eux-mêmes cette coopération ; l'hypothèse que les utilisateurs doivent fournir des définitions précises pour les modèles d'activités n'est pas viable (interaction par spécification). Nous espérons avoir mis en lumière par ce présent travail le problème critique de la coopération de systèmes flexibles et de pouvoir ainsi favoriser de nouveaux travaux sur cette problématique.

En fixant des contraintes à respecter dans ce nouveau type d'interopérabilité : visibilité, flexibilité, nous fixons des premières lignes conductrices à suivre pour les futurs travaux sur ce problème. Par la suite, ces travaux seront à enrichir sur les aspects d'interopérabilité politique, humaine et légale (cf. chapitre 2). Pour l'instant, nous avons fait le choix de proposer une solution générique, c'est-à-dire qui permettent tout type de coopération concernant ces précédents aspects. Néanmoins, des mécanismes énoncés par des travaux en rapport avec ces niveaux plus élévés pourraient être intégrés à CAST.

Il est important de rappeler que CAST ne fournit pas une passerelle complète (de coopération) entre deux systèmes, mais une majeure partie de celle-ci. L'implémentation de la passerelle pourra être complétée par des aspects plus "systèmes" (ex : aspects transactionels) si ces derniers doivent rentrer en compte dans la coopération.

2.2 Transformation de modèles, interopérabilité et Design Pattern

Nous avons montré que les transformations de modèles ne sont pas adaptées à ce problème. Alors qu'elles sont souvent mises en avant, nous avons démontré qu'elles ne sont pas une réponse à tous les problèmes d'interopérabilité. La flexibilité exige une dynamicité que les transformations ne proposent pas. Le problème peut-être amoindri avec une couche événementiel (comme pour le CWM) mais 1) il manque encore des mécanismes de répartition (géographique) des tâches 2) les caractéristiques dérivées ne peuvent être transformées 3) la couche événementielle vient diminuer le degré d'autonomie.

Nous avons apporté aussi un formalisme de liaison d'adaptation qui propose une vision différente de la transformation. Par rapport à CAST lui-même, nous avons voulu surtout montrer que la coopération de systèmes flexibles est plus une affaire d'adaptation que de transformation simple. L'adaptation est au départ une transformation, mais elle implique une association forte avec la source et une vie réelle de la transformation. L'adaptation permet en plus la répartition des tâches (l'aspect "coopération" est vraiment présent). Mettre en avant de tels avantages pour une interopérabilité sémantique est notre apport le plus important. Les adaptateurs servent souvent à un niveau très technique : protocole de communication, intergiciel (ex : Corba), partage de ressources... Ici nous avons montré que leur utilité est tout aussi grande à un haut niveau d'abstraction.

Sur la conception même des adaptateurs, nos travaux constituent une nouvelle technique de construction de tels objets. Les trois points suivants montrent cette évolution :

- Les règles d'adaptation : si, comme certains intergiciels (Corba grâce au DSI), CAST fournit un moteur de génération d'adaptateurs pouvant contenir tout type de traitement/re-direction, la re-direction est construite à partir de règles d'adaptation. L'emploi de règles accélère l'implémentation de la re-direction.
- La flexibilité : la structure des adaptateurs permet une modification du code de re-direction à tout moment, sans arrêt de l'exécution et toujours à partir des règles d'adaptation.
- La "bi-directionnalité" : L'héritage multiple permettait, à l'inverse du surcouchage, d'avoir des adaptateurs bidirectionnels. Comme cette technique posait différents problèmes, nous avons amélioré la technique du surcouchage pour qu'elle devienne elle-aussi bidirectionnelle. Cela peut être translaté dans d'autres contextes et augmenter ainsi les possibilités de réutilisation de code d'implémentation.

2.3 Favoriser la méta-modélisation et la structuration en trois niveaux

RAM3 est, à notre avis, un outil qui rend plus accessible la méta-modélisation. Dans le chapitre 6, nous avons déjà souligné que la méta-modélisation est un exercice mental difficile. La modélisation n'est déjà pas un exercice facile. L'instanciation et l'intercession sont deux aides à ce processus. L'instanciation permet d'accéder à la réalité que décrit un modèle, de "palper" ses éléments, et de vérifier ainsi que la structure et le comportement de ces derniers ne comportent pas de lacunes ou de surplus. Des mécanismes d'intercession permettent ensuite d'ajouter ou de supprimer certaines caractéristiques sur les instances déjà créées et de voir ainsi le résultat des modifications. Avec la méta-modélisation, un niveau d'abstraction supplémentaire apparaît, et la difficulté de définition augmente. Lorsque l'on crée un nouveau méta-modèle, pour des raisons scientifiques (comme DARE) ou industrielles (comme le workflow), il est difficile de cerner la totalité du domaine de réalité issu de ce méta-modèle. Encore une fois l'instanciation et l'intercession permettront de voir quels types de modèles il est possible de créer et de modifier le méta-modèle si les possibilités de création sont insuffisantes ou inappropriées. Le niveau d'abstraction supplémentaire implique un besoin de double instanciation et double intercession : le modèle que je viens de créer (avec ce nouveau méta-modèle) est-il juste ? Il me faut pouvoir l'instancier. Si, grâce à cette instanciation, je vois que le modèle ne convient pas, est-ce une erreur de ma part dans la définition du modèle ou dois-je changer le méta-modèle pour ajouter des éléments au modèle ? Dans le dernier cas, si je rajoute une caractéristique à une entité du méta-modèle, quelle est sa conséquence sur le précédent modèle, et sur les précédentes instances de celui-ci ? La double instanciation et la double intercession de RAM3 sont donc un atout de poids pour la méta-modélisation et facilitent ainsi son apprentissage.

L'opérationnalisation d'un méta-modèle est l'autre problème majeur en ce qui concerne la méta-modélisation. La liste, présentée dans le chapitre 4, des mécanismes à implémenter pour l'opérationnalisation d'un méta-modèle montre la difficulté de celle-ci. Après la représentation mentale peu aisée, c'est la seconde difficulté qui peut rebuter un développeur à adopter une approche par méta-modèle. En fournissant directement un serveur de données (à trois niveaux), RAM3 écarte cette seconde difficulté et favorise ainsi la précédente adoption.

Adopter une approche par méta-modèle peut sembler parfois inutile par rapport à l'utilisation d'un langage orienté objet et de la spécialisation[84]. Dans cette alternative, la définition du concept de tâche de DARE revient à définir une classe *Task* et à spécialiser celle-ci pour chaque nouvelle tâche (ex : *CoursA*, *TPA*). Le problème du choix (ou de la proximité) entre instancier et spécialiser est récurrent : d'un point de vue conceptuel, le cours A est-il une tâche (instanciation) ou une sorte de tâche (spécialisation) ? La méta-modélisation (par rapport à la spécialisation de classes) a comme particularité de permettre des capacités de modélisation réduites et de guider ainsi le concepteur dans la définition d'un modèle (et de l'empêcher de sortir du domaine d'application prévu). Dans le cas où les méta-modèles sont très orientés vers un domaine précis, les modèles ont pour second avantage d'être plus compacts. Nous espérons qu'avec RAM3, les difficultés d'opérationnalisation ou de représentation mentale ne viennent plus interférer dans le précédent choix (méta-modélisation ou spécialisation).

Enfin une grande originalité de notre approche est le fait de factoriser une certaine construction (i.e. des passerelles pour la coopération) grâce à la particularité de la structuration en trois niveaux (concepts de type et d'instance, lien d'instanciation, caractéristiques répercutées). Nous montrons ainsi comment bénéficier de ce type d'architecture.

2.4 Le contexte nouveau du Model Driven Architecture

2.4.1 Contexte

Le MDA est une étape importante dans l'histoire du génie logiciel. Depuis de nombreuses années, les recherches dans ce domaine encouragent, dans le conception d'une application, l'utilisation de formalisme de modélisation abstrait : définir l'architecture de l'application sans tenir compte des préoccupations d'implémentation. Cette approche plus conceptuelle permet généralement d'avoir un programme final plus clair, plus compact et plus évolutif. La création d'UML et la politique d'expansion de l'OMG vis-à-vis de celui-ci ont déjà fait évoluer une bonne partie des développeurs. Pourtant un certain nombre d'entre eux continuent de développer une application directement à partir d'un langage de programmation (ou de RAD). En définissant le MDA comme sa nouvelle architecture normalisée, l'OMG tourne la page définitivement : le concepteur d'une application n'a plus à se préoccuper du tout de l'implémentation. L'idée principale du MDA est de modéliser une application à l'aide d'un méta-modèle MOF ou d'un profil UML métier (c'est-à-dire adapté au domaine de l'application) et de choisir la plate-forme d'implémentation. Des règles de transformation associées au méta-modèle ou au profil précédent font le reste. L'architecture de transformation du CWM est reprise comme base des règles de transformation du MDA. Le vote en faveur de l'adoption du MDA est récent (fin de l'année 2001). L'OMG se laisse quelques années pour l'adoption complète. Ce laps de temps a aussi pour objectif de dresser une liste des possibilités de conception grâce à cette approche. Les sources "d'inspiration" de cette liste sont, comme à l'habitué, les différentes RFI (*Request For Information*) et autres TF (*Task Force*) de l'OMG.

2.4.2 Différentes propositions

Différentes propositions sont donc apparues vis-à-vis du MDA. Une liste exhaustive serait hors de propos. Les deux propositions suivantes résument assez bien à notre avis les deux courants principaux.

Flater dans [FLA 02] indique que les communautés propres à chaque domaine (Santé, aérospatial, télécommunication, ...) seront encouragées à se réunir pour définir des méta-modèles métiers, c'est-à-dire propre à leur domaine (comme le récent méta-modèle d'EML [KOP 02] est propre à la pédagogie). Chaque domaine devrait ainsi avoir son méta-modèle "standard". Chaque communauté définira ensuite les règles de projection vers au moins un support/méta-modèle d'implémentation (EJB, Corba, .net, ...). Un concepteur souhaitant développer une application

[84] Il ne s'agit pas ici du choix du patron de conception pour l'implémentation d'un méta-modèle.

destinée à une société de télécommunication utilisera le méta-modèle associé pour modéliser son application et le moteur de transformation générera l'application finale. L'OMG insiste pour évoquer la nécessité du concepteur à produire les 5 ou 10 % de code restant à effectuer : tout ne peut être généré automatiquement.

À travers Pegamento, Raymond dans [RAY 01] propose un nouveau type d'outil de développement. Son originalité est d'être orienté modèles. L'idée centrale est toujours de partir d'un méta-modèle très abstrait et très spécifique à un domaine particulier et de finir sur un méta-modèle d'implémentation. Par contre, le chemin peut-être plus long que dans l'hypothèse de Flater. Il peut y avoir plusieurs méta-modèles de plus en plus spécifiques à la plate-forme d'implémentation, comme des méta-modèles spécifiques aux interfaces graphiques ou à la sécurité. La notion de *raffinage* est très présente dans cet outil. Les règles de transformation peuvent, en plus, être modifiées grâce à l'outil. La nécessité de normaliser un méta-modèle par domaine de la première proposition n'apparaît pas du tout ici. Il s'agit plutôt de développer une application en évitant le plus possible d'utiliser un langage de programmation grâce à l'utilisation de méta-modèles spécifiques.

2.4.3 Les extensions M-zéro

L'objectif principal du MDA en ce qui concerne son utilisation est la construction d'applications. Dans nos travaux, nous avons surtout abordé la construction, l'architecture et le fonctionnement des "systèmes". L'application peut être considérée comme un sous-ensemble d'un système qui lui est plus générique : le système se base sur un méta-modèle alors qu'une application se base sur un modèle. De ce fait les extensions que nous avons définies sont peu utiles : lorsque le concepteur utilise un méta-modèle, celui-ci ne crée que des modèles et pas d'instances de modèles. Les modèles sont ensuite traduits en autres modèles. Il s'agit cette fois de modèles d'implémentation qui eux vont être instanciés. Si nos extensions ne sont pas intégrables au MDA la base conceptuelle sur laquelle elles reposent est par contre beaucoup plus pertinente. Sur l'exemple du méta-modèle DARE qui servirait à décrire des applications coopératives, comment pourrait se traduire une tâche en Java ? La tâche *TPA* deviendrait une classe Java TPA héritant d'une classe Java *Task*. Elle serait liée à une classe *Etudiant* (qui hérite de la classe *Role*) par un attribut *etudiants*. C'est la notion de points d'entrée secondaires qu'on retrouve ici. La correspondance *Dare.Task* → *Java.Class* doit amener à se questionner sur la capacité de la relation Class⇔Object à implémenter la relation CT⇔CI. La nécessité de cardinalités secondaires dans un modèle devient ici critique : combien d'étudiants peut-il être ajoutés ?

Nos travaux sur la définition des liens entre concepts de type et concepts d'instance peuvent, à notre avis, servir à factoriser la création de règles de projection vers un support d'implémentation, c'est-à-dire répondre à la question "quels sont les éléments qui interviennent généralement dans la projection vers un support d'instanciation/implémentation ?"

Dans la génération de systèmes, c'est-à-dire l'opérationnalisation d'un méta-modèle, les extensions M-zéro restent par contre aussi importantes.

3 Poursuite des travaux

Cette partie présente les perspectives de recherche liées au développement de CAST et RAM3. Ces perspectives s'insèrent dans la communauté du génie logiciel (GL), dans celle du TCAO/ACAO (T/A) et celle plus technologique du WEB (WEB). La présentation se découpe en quatre parties qui correspondent chacune à un possible projet : la construction d'un outil de meta-case puissant (GL), la validation et l'évolution de CAST (GL), l'amélioration de DARE grâce à RAM3 (T/A) et enfin l'application de RAM3 et CAST à la plate-forme web Réciprocité (T/A et WEB).

3.1 Un outil de meta-case puissant (GL)

RAM3 constitue un outil de méta-modélisation MOF avec des capacités de prototypage importantes. Notre idée pour l'évolution de RAM3 est d'en dériver un puissant outil de meta-case, c'est-à-dire un outil de conception avec un haut degré de personnalisation. La personnalisation serait basée sur l'emploi de la méta-modélisation, d'où le nom provisoire de M3CT (Meta-Modeling for Meta

Case Tool). Cette évolution pourrait s'inscrire dans le cadre d'un projet en partenariat avec une société spécialisée dans le domaine des AGLs, par exemple MEGA [MEG 02] qui propose une solution d'EAI (*MEGA Integration*), mais surtout de puissants outils de modélisation UML (*MEGA Development*) ou de modélisation de processus (*MEGA Process*).

La première section est une vue générale qui reprend toutes les caractéristiques de ce futur outil. Chacune de ces caractéristiques est l'objet d'une des sections suivantes.

3.1.1 Vue générale

L'objectif premier de M3CT est la conception. Il y a deux possibilités : la conception d'application et la conception de système. Tout au long de cette partie, les fonctionnalités de M3CT sont décrites dans le contexte de construction d'une application. Celles-ci sont néanmoins reprises, dans la dernière section, dans le contexte de la conception d'un système.

Méta-modèle personnalisé - La conception d'une application avec M3CT débute par le choix du méta-modèle. Le formalisme de définition des méta-modèles est bien sûr le MOF. Le concepteur indique dans quel formalisme, il souhaite définir le modèle de l'application. Le méta-modèle choisi peut être un méta-modèle existant (chargé à partir d'un document XMI) ou créé à partir de M3CT. Le concepteur définit ensuite un modèle. Les capacités de prototypage de M3CT issues de RAM3 permettent au concepteur un affinage très efficace du modèle. **Notation UML et support XMI** - La création d'un méta-modèle est faite à partir de la notation UML et le chargement de méta-modèles ou de même modèles (créés à partir d'autres outils – quelque soit le méta-modèle dont ils sont issus) est réalisée grâce à un support XMI complet. **Association d'artefacts graphiques** - Pour tout méta-modèle chargé dans M3CT, le concepteur peut associer aux concepts et aux associations du méta-modèle des artefacts graphiques afin de pouvoir modéliser graphiquement. La modélisation graphique est plus rapide et plus intuitive que celle textuelle ou celle plus évoluée de RAM3. **Méta-modèles additionnels** - Parce que différents aspects interviennent dans une application, il est intéressant de modéliser chacun de ces aspects à partir d'un méta-modèle approprié. À cet effet, M3CT propose d'associer des méta-modèles à un méta-modèle central en tant que vues. Ces méta-modèles additionnels sont liés au méta-modèle central par des liaisons d'adaptation. Cette modélisation à vues multiples permettra à plusieurs concepteurs (de spécialité différente) de travailler ensemble simultanément. **Transformation de modèles** - Pour la réutilisation de modèles anciens ou l'échange de modèles entre AGLs différents, M3CT fournit un outil de script et un éditeur de (liens de) transformation. Ces deux fonctionnalités permettent au concepteur d'effectuer facilement des transformations de modèles. **Génération de code** - Ce type d'opérations permettra d'ailleurs à générer du code d'implémentation.

3.1.2 Méta-modèle personnalisé

Il est bien évident que selon le domaine de l'application, des méta-modèles seront plus adapté que d'autres pour la modélisation de celle-ci. Par exemple pour définir des activités coopératives, le formalisme de DARE est plus adapté que le formalisme UML. Le principal problème des outils de modélisation est de ne proposer qu'un seul formalisme de modélisation ou pour les outils de meta-case qu'un nombre non extensible de formalismes. Un concepteur spécialisé dans un domaine peut souhaiter créer son propre formalisme de modélisation afin d'en disposer d'un plus adapté, c'est-à-dire plus rapide dans la description et plus compréhensible dans la relecture. Il ne s'agit pas de favoriser la création d'une multitude de méta-modèles réduisant ainsi les échanges entre AGLs à zéro mais plutôt de permettre aux concepteurs, à travers la personnalisation de méta-modèles, de modifier leur environnement de travail pour l'adapter à leur expérience et à leurs nouveaux besoins[85]. Des outils de transformations de modèles (discutés ultérieurement) permettent de régler le problème des échanges entre AGLs.

[85] La modélisation est une activité. Selon la Théorie de l'Activité, son contexte doit évoluer.

Un des objectifs de M3CT est donc que le concepteur puisse modéliser sa future application à partir de n'importe quel méta-modèle. M3CT peut créer de nouveau méta-modèle mais aussi en importer/charger de nouveaux grâce au support XMI. Le moteur (d'exécution) à quatre niveaux de RAM3 à la base de M3CT permet à celui-ci de créer des modèles de tout types de méta-modèles et d'instancier ces modèles. Cette dernière caractéristique permet au concepteur d'instancier son modèle (de l'application) et de vérifier sur les instances le bien-fondé de ses choix de modélisation. Le moteur réflexif de RAM3 apporte la possibilité de modifier le modèle et d'en voir les répercussions sur les instances déjà existantes. Avec l'association éventuelle de code Java au modèle par le concepteur, ce dernier peut disposer d'instances plus "vivantes". L'exécution de celle-ci aide de nouveau le concepteur à évaluer la correspondance de son modèle à ses besoins.

M3CT offre donc la possibilité au concepteur de modéliser son application à partir d'un méta-modèle (existant ou créé) adapté à celle-ci et le guide dans sa modélisation grâce à ses capacités de prototypage.

3.1.3 Notation UML et support XMI.

La section 1.3.2.2 explique l'avantage d'utiliser la notation UML pour la définition de méta-modèle plutôt que d'employer les éditeurs de RAM3. UML est plus rapide pour la définition de méta-modèles MOF et se trouve être le formalisme standard pour la précédente opération. Pour ces mêmes raisons, M3CT intégrera la notation UML pour la définition de méta-modèle.

Pour rester aussi dans la norme, un support XMI complet sera intégré dans M3CT. Cette perspective est très intéressante. Premièrement, il permet de charger des méta-modèles définis par d'autres outils MOF (Rational Rose, dMOF, M3J, XMI Toolkit, ...) mais aussi des modèles UML pour les retravailler dans l'environnement M3CT (transformation, vues multiples). Le support XMI offre aussi la possibilité de charger des modèles de n'importe quel méta-modèle (et donc de n'importe quel système à trois niveaux ou AGL respectant la norme XMI). Ces modèles pourraient être eux aussi retravailler avec M3CT et le résultat sauvegardé toujours en XMI (donc ré-injectable dans le système ou AGL d'origine).

3.1.4 Association d'artefacts graphiques

Un des inconvénients majeurs des rares outils (les sNets, RAM3, TokTok[86] [STE 01][TOK 02], MetaGen) qui permettent une modélisation à n'importe quel méta-modèle est la faiblesse des interfaces graphiques. La modélisation est soit textuelle (RAM3 ou TokTok) soit graphique (sNet/Semantor, MetaGen) mais dans ce cas, le formalisme graphique est le même quelque soit le méta-modèle utilisé. Pour les sNets, on manipule toujours des ronds et des flèches. Il n'est pas possible par exemple d'associer au méta-modèle DARE le formalisme graphique présenté dans le chapitre 4 (rectangle, rectangle aux coins arrondis et personnage). Pourtant un artefact graphique spécifique à un concept ou une association d'un méta-modèle constitue une accélération dans l'utilisation de ce dernier. Par exemple, la définition d'une composition en UML se définit plus facilement à travers le placement d'un trait terminé par un losange plein (Rational Rose) qu'à travers les menus contextuels de RAM3. L'utilisation d'artefacts graphiques spécifiques est plus rapide et plus intuitive que l'emploi de notation textuelle, de navigateurs ou de formalisme graphique général.

Notre objectif est donc de permettre aux concepteurs de pouvoir créer et associer un formalisme graphique à chaque méta-modèle. Ils disposeront ainsi de réels éditeurs graphiques de modèles[87]. Le module de M3CT pour la création d'un formalisme graphique et de son association à un méta-modèle est la composition des éléments suivants :

---

[86] TokTok est un outil associé à dMOF qui permet de définir de manière textuelle des modèles quelque soit le méta-modèle.

[87] La notation UML utilisé pour la définition de méta-modèles sera elle-même le produit d'un ensemble de correspondances entre concepts/associations UML et artefacts graphiques.

- Un ensemble d'artefacts graphiques de base (rond, rectangles, losange, trapèze, parallélépipède, clé à molette, roue à pignon, personnage, ...).
- Un éditeur de dessin vectoriel dont les fichiers de sauvegarde respectent une norme connue (ex : eps, wmf, ...). Un tel support permet au concepteur d'importer des dessins déjà existants ou de créer des dessins avec des outils plus puissants si le concepteur en possède.
- L'éditeur de correspondance entre artefacts graphiques et concepts/associations du méta-modèle.
- Un placeur de textes. Ces textes correspondent aux propriétés définis par les concepts (ex : le carré est associé au concept de Tache, placer le commentaire en bas à droite).

La principale difficulté provient de l'éditeur de correspondance et plus particulièrement des associations (la correspondance artefacts-concepts n'est pas problématique) : par exemple faire correspondre la flèche avec un losange plein avec l'association de composition. Pouvoir créer tout les types de flèches possibles n'est déjà pas une chose aisée. Mais elle se révèle à notre avis plus facile à définir que le second type de liaison graphique : l'embriquement. Par exemple, dans le méta-modèle EJB++ d'O. Caron [CAR 02] (issu des EJBs et du CORBA Component model [OMG 02b]), le lien graphique entre le réceptacle et la facette est une juxtaposition, c'est-à-dire un type d'embriquement. La figure 96 montre plusieurs types d'embriquement.

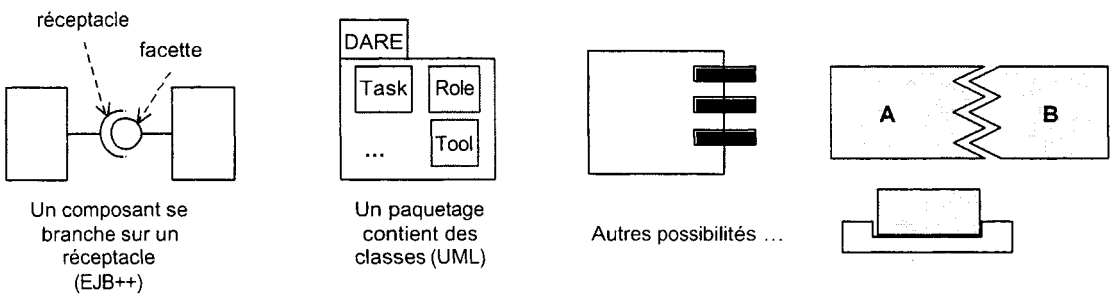


figure 96. Exemples d'embriquement

La correspondance artefact-association est donc plus complexe que la correspondance artefact-concept. Dans nos perspectives concernant le développement d'une première version de M3CT, nous limiterons sûrement le module de correspondance artefact-association à un couple un éditeur de 'flèches' et à une batterie de types d'embriquement (cette batterie pourrait être extensible à partir de module de code). Cette première version nous fournira une base pour expérimenter la création de formalismes graphiques et ainsi peut-être établir des règles de factorisation concernant la définition d'embriquement.

3.1.5 Méta-modèles additionnels : plusieurs vues possibles pour un même modèle

Lors de la modélisation d'une application, plusieurs aspects rentrent en jeu : la partie fonctionnelle, la partie sécurité, la partie IHM, la partie déploiement, ... Disposer de moyens de définitions spécifiques à chaque aspect permet une modélisation plus cohérente, plus claire et facilite le travail coopératif de plusieurs concepteurs aux spécialités différentes. Vis-à-vis du ou des méta-modèles employés pour la modélisation de l'application, plusieurs 'configurations' peuvent exister :

1. **Des concepts très fournis** : Un seul méta-modèle est utilisé et ses concepts possèdent les caractéristiques nécessaires pour définir les aspects de sécurité, d'interface utilisateur ... Ceci implique pour chaque concept une liste des caractéristiques assez longue. Les concepts sont de ce fait plus complexes. Le problème de nommage, des adaptateurs de type

fusionnel, évoqué dans le chapitre 4 (section 3.2) est susceptible d'apparaître – des caractéristiques de même nom (ex : *commentaire*). Des noms allongés (ex: *commentaire DeDistribution*) offrent une solution basique mais très efficace. De plus, les interfaces utilisateurs ne montrent pas les noms complets (les noms raccourcis restent suffisamment désignatifs grâce à leur contexte d'utilisateur – ex: l'éditeur de distribution n'affiche que *commentaire*).

2. **Une composition d'aspects autour de concepts centraux** : Les travaux de R. Marvie sur la construction d'application à base de composant adoptent une approche de 'type MDA'. L'originalité de ces travaux provient d'un outil basé sur le méta-modèle CCM qui permet de spécifier des informations additionnelles aux modèles. Le concepteur peut 'brancher' d'autres éléments de modélisation, issus d'autres méta-modèles, sur son modèle d'application décrit en terme de composants. Le concept de composant est donc associable à d'autres concepts dont l'objectif est de décrire un aspect particulier. L'outil est encore en cours de conception tout comme le moyen de définition des méta-modèles connexes. Un problème délicat de cette approche est à notre avis la présence d'éléments de définition qui soient en opposition : par exemple, l'incohérence peut venir du modèle de sécurité qui implique que les deux éléments A et B doivent être sur le même poste et du modèle de déploiement qui indique que A et B peuvent être situés sur des postes différents.

3. **Un raffinage du modèle à travers des transformations successives** : Le développement d'une application est une transformation successive d'un modèle : modèle d'analyse → modèle de conception → modèle d'implémentation. Des outils comme Rational Rose effectuent déjà des transformations du modèle de conception en un modèle d'implémentation. L'objectif de Pegamento [RAY 91], déjà présenté en 2.4.2, est, en partie, de générer des outils de transformation de modèles. Pegamento peut alors servir pour assister la transformation analyse → conception. Pegamento pourrait servir à agrandir la chaîne de départ (analyse → ... implémentation) : la modélisation débuterait toujours par une description des tâches (méta-modèle d'analyse) et se terminerait par un modèle d'implémentation mais des méta-modèles transitoires feraient apparaître chacun un des précédents aspects (sécurité, IHM, déploiement, ...). Le principal problème dans cette dernière hypothèse est à notre avis les répercussions entre aspects lors des transformations. Si les aspects de déploiement sont modélisés après ceux de sécurité, ces derniers sont peut-être altérés et nécessitent une re-définition. Comme les répercussions n'apparaissent qu'à la transformation, le travail coopératif de plusieurs concepteurs (avec chacun une spécialité) n'est pas rendu facile. Parce qu'apporter des mécanismes de coopération dans la conception d'applications nous semble important, nous sommes plus enclins à adopter une approche plus dynamique (cf. la proposition suivante).

→ **Des méta-modèles additionnels comme vues** : La création de vues grâce à des méta-modèles additionnels est l'approche que nous souhaitons outiller. Cette proposition vise à apporter une plus grande coopération entre les concepteurs (problème des répercussions – point 3), à pouvoir ajouter différents aspects de la conception (non extensible en 1), à ne pas avoir des méta-modèles trop fournis et donc trop complexes (point 1), à ne pas avoir de problème de nommage (point 1) et à ne pas d'incohérence (point 2).

Le principe est d'avoir pour base un méta-modèle général comme UML (pour l'échange avec d'autres AGLs) ou un méta-modèle d'implémentation (Java/EJB, CCM, .Net ...). Sur celui-ci viennent se greffer des méta-modèles additionnels : chacun des méta-modèles est lié au méta-modèle de base par des liaisons d'adaptation. L'objectif de ces liaisons n'est pas la jonction de deux référentiels de modèles mais l'accès au référentiel (de base) par un méta-modèle différent. La jonction d'un méta-modèle correspond à la création une vue. Par rapport à la programmation orientée aspect ou les vues multiples des SGBDs, on dispose ici d'un niveau méta supplémentaire car la jonction d'un méta-modèle implique des liens d'adaptation entre concepts de type mais aussi entre concepts d'instance : en plus des modèles visibles de point de vues différents, leurs instances aussi, créées pour les tests, peuvent être manipulées à travers ces mêmes points de vue. La conception d'une telle fonctionnalité

(méta-modèles additionnels) "n'est qu'une dérivation" de l'outil M-CAST présenté dans le chapitre 6 : il s'agit d'en faire une version mono-référentiel.

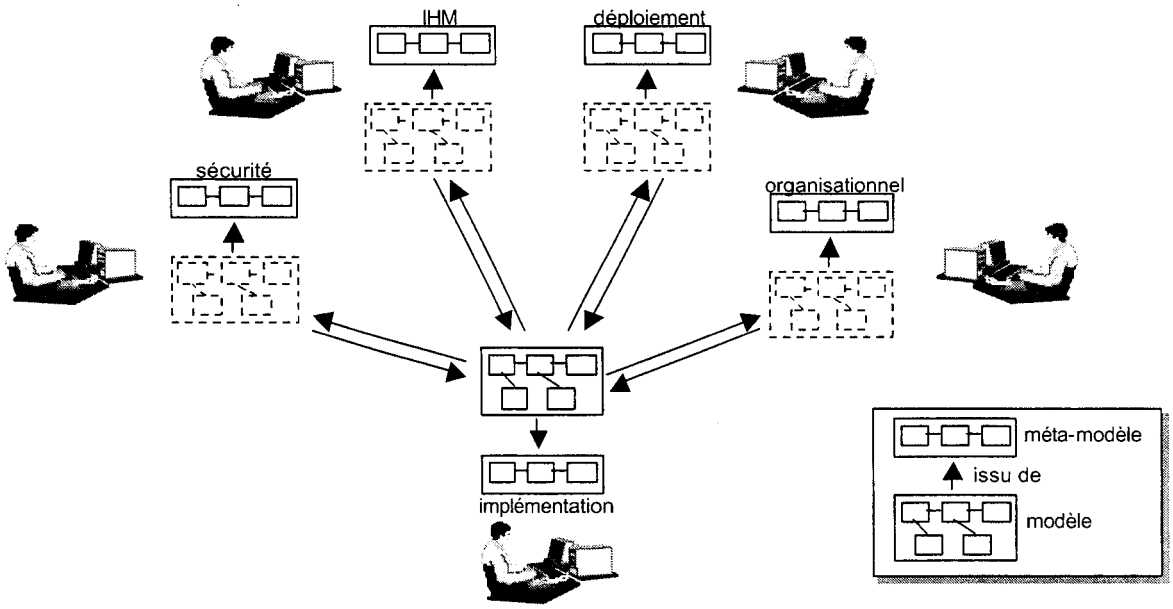


figure 97. Exemple de vues mu ltiples

Cette architecture permet à plusieurs concepteurs (chacun sur un poste différent) de travailler sur le même ensemble d'éléments (de modélisation) à partir de méta-modèles différents. De tels mécanismes de partage apportent une grande dimension de coopération. Si la personne en charge du modèle de sécurité apporte une modification de son modèle, il ne fait que modifier le modèle de base. Pour cette raison, chaque concepteur voit automatiquement son modèle (i.e sa vue du modèle de base) se modifier. Comme souvent dans les espaces de partage, une fonctionnalité importante qu'il faudrait implémenter serait la pose de verrous.

Notre idée n'est pas de fournir un méta-modèle de base en particulier avec les différents aspects/méta-modèle pour la conception d'applications mais plutôt un outil pour la création d'une telle configuration. Une fois encore, le concepteur personnalise son environnement : il choisit un méta-modèle de base et lui associe les méta-modèles/vues. Notre priorité est d'offrir une grande liberté aux concepteurs. Des règles de transformation du méta-modèle de base vers un méta-modèle standard comme UML peuvent aussi être définis dans le cadre d'échange avec d'autres AGLs (transformation évoquée plus loin). L'usage simultané des méta-modèles additionnels associés et du module de correspondance d'artefacts graphiques procurerait un outil de modélisation doté d'un grand potentiel.

3.1.6 Transformation de modèles

La transformation de modèles est une opération très utile dans la conception d'applications. Elle permet par exemple de réutiliser tout ou partie d'anciens modèles d'applications (c'est-à-dire issus de méta-modèles anciens – comme l'exemple Merise/UML du chapitre 5) ou d'échanger des modèles entre AGLs (et donc entre équipes de développement différentes). Il est donc important que M3CT propose des outils pour la transformation de modèles.

Nous envisageons d'en proposer deux. Les fenêtres de script de RAM3 dont est issu M3CT fournissent le premier outil. Nous en avons déjà décrit une telle utilisation dans le chapitre 6. Le second outil proviendrait d'une évolution de CAST. L'éditeur CAST étant proche du paquetage

Transformation du CWM, il est logiquement bien adapté à la transformation de modèles[88]. L'évolution de CAST consiste à modifier le moteur d'exécution de celui-ci (i.e le paquetage *CAST*) pour qu'il effectue, selon les règles définies, des transformations de modèles et non une génération de paquetages *X_Adapter* et *X_Filter*. Les interfaces de l'éditeur de CAST ne changent que très peu. C'est principalement l'état d'esprit d'utilisation qui change. Les fonctions de transformation sont conçues différemment des fonctions d'adaptation, il s'agit d'une transformation définitive/statique et non temporaire/dynamique : A→B ne se lit pas *B se déduit de A* mais *A donne B*. C'est pour cela que les fonctions (d'adaptation) d'accès *add*, *modify*, *set*, ... sont ici inutiles et ne seraient plus présentes dans une telle évolution de l'éditeur CAST (c'est le principal changement au niveau des interfaces de celle-ci).

3.1.7 Génération de code

Tout AGL qui propose une génération de code à partir d'un modèle créé apparaît plus attractif pour les concepteurs (la modélisation souvent vue comme facultative par ces derniers se révèle alors comme un catalyseur dans la construction globale). M3CT se doit de proposer une génération de code à partir des modèles créés. Néanmoins M3CT permet de travailler tout type de méta-modèle. Et dans les AGLs, la génération de code est rattachée à un méta-modèle en particulier. Dans Rational Rose, il s'agit de transformer un modèle UML en code Java, Visual Basic, C++ ou interfaces IDL. La génération ne fonctionne qu'avec un seul méta-modèle. À partir du moment où M3CT travaille avec une infinité de méta-modèles, le problème de génération prend une autre dimension. Nous n'avons pour l'instant pas encore approfondi la question. Notre idée de départ est de travailler sur un outil pour créer des règles de génération de code et ceci à partir d'une spécialisation de l'outil de transformation (évolution de CAST). Cette spécialisation serait destinée à tracer des liens de transformation de n'importe quel méta-modèle vers le méta-modèle Java (dans un premier temps). Le moteur d'exécution (des règles) de cette spécialisation générerait (à partir de modèles divers) des modèles Java traduits ensuite par un module (à créer) en programmes Java. La spécialisation n'a pas comme seule particularité le module de traduction en programmes Java (qui pourrait exister indépendamment). La principale motivation de la spécialisation serait de faire évoluer les interfaces utilisateurs (de l'éditeur de liens de transformation) pour que celles-ci deviennent propres à la génération de code. Par exemple, dans la cas où le concepteur travaille sur les règles de génération à partir du méta-modèle de DARE, la liaison du concept *Task* pourrait débuter par une proposition de l'éditeur (spécialisé) de génération vers différents types de classes pré-définies (classes avec fenêtres graphiques, classes avec persistances, classes héritant d'une autre, ...). L'objectif des interfaces serait, en plus de la définition de règles de transformation, de fournir un assistant spécifique à la génération. Des mécanismes d'extension de ces interfaces utilisateurs seraient la deuxième étape de notre recherche.

3.1.8 Développement de système (support à un système d'information)

Nous avons décrit les (futures) fonctionnalités de M3CT pour la conception d'applications. M3CT est bien sûr utilisable pour concevoir des systèmes, mais son utilisation devient différente lorsqu'il s'agit de concevoir un système structuré en trois niveaux. Les fonctionnalités précédentes font alors partie intégrante du système développé et le concepteur se focalise surtout sur le (ou les) méta-modèle(s) du système. Les modèles et leurs instances sont juste des vecteurs de test du (ou des) méta-modèle(s) nouvellement défini(s).

Tout comme un système construit grâce à RAM3 repose sur une version allégée de celui-ci à l'exécution, il en est de même pour un système construit à partir de M3CT, c'est-à-dire que le système se repose sur le moteur à quatre niveaux et peut utiliser les interfaces utilisateurs (avec artefacts graphiques), les méta-modèles additionnels, le moteur de transformation et les fenêtres de script. Le reste de l'implémentation du système peut être développé avec M3CT en tant qu'application. Cette partie restante utilisera l'éditeur de modèles construit à partir de M3CT. L'éditeur sera complètement graphique (grâce à la correspondance d'artefacts graphiques). Il pourra proposer plusieurs vues

[88] Les liaisons se définirait cette fois sur des modèles et non des méta-modèles comme pour CAST.

possibles, correspondantes chacune à un méta-modèle particulier. L'accès à ces vues pourra être synchronisé avec le profil de l'utilisateur. La partie de l'implémentation pourra enfin utiliser l'API de M3CT pour concevoir des fenêtres de script personnalisées ou utiliser directement celles de M3CT.

RAM3 et CAST fournissent une base solide pour l'implémentation des fonctionnalités présentées dans cette partie. La perspective de disposer de toutes ces fonctionnalités dans un même outil de conception, M3CT, fournirait un outil de meta-case très étoffé, coopératif, et proposant un degré de personnalisation jamais atteint.

3.2 CAST

Les travaux de CAST n'ont pas fait partie d'un projet co-financé par un contrat de recherche. Jusqu'à présent notre approche a été conceptuelle. La phase de validation de CAST serait indéniablement mieux réalisée sur un cas réel de coopération, c'est-à-dire où les acteurs du partenariat ont de réelles attentes vis-à-vis de la coopération. CAST peut fournir une coopération entre deux organisations qui nous semble correct mais qui ne convient pas aux besoins spécifiques d'une entreprise et à son utilisation quotidienne.

Le principal problème de la validation de CAST est l'absence de système utilisant le MOF. Pour cette raison, il nous faut proposer un outil qui permette de créer rapidement une exportation MOF pour des systèmes déjà existants. Il nous faut aussi favoriser l'implémentation de la structuration en trois niveaux, pour les systèmes à venir. La validation nécessitera bien évidemment d'achever l'implémentation de l'éditeur de liaison d'adaptation et du module de génération mais aussi de peut-être élargir CAST à d'autres supports (de modèles) que le MOF. Une des perspectives d'évolution de CAST est la possibilité de créer des langages pivot dans le cadre d'une fédération de systèmes.

3.2.1 Système à trois niveaux et le MOF

Nous avons vu dans le chapitre 3 que la structuration en trois niveaux est utilisée dans des meta-groupware récents et sera à notre avis de plus en plus adoptée. L'approche CAST trouve donc des systèmes sur lesquels elle peut être appliquée. La situation est différente pour M-CAST car il n'a pas encore de systèmes (à trois niveaux) utilisant des référentiels de modèles MOF. Cette situation est la raison pour laquelle un des objectifs de RAM3 (dans la perspective de validation de CAST) est de disposer d'interfaces utilisateurs spécifiques à la création d'une couche d'exportation MOF pour des systèmes déjà existants. Ces interfaces seraient essentiellement spécifiques à des supports possibles de modèles, par exemple DOM+XML (non XMI), Java/EJB, SGBDR, etc. Il s'agirait de déduire comment le support a été utilisé pour décrire des modèles et leurs instances (problème étudié dans le chapitre 3). Pour les référentiels construits avec un SGBDR on peut imaginer une interface de saisie des tables existantes, et des interfaces de correspondances entre entités/tables, attributs/champ, ... Ces interfaces produiraient les requêtes SQL (ou autres) correspondantes.

Même si des systèmes architecturés en trois niveaux existent, ils ne sont toutefois pas nombreux. Il peut y avoir plusieurs raisons à cela :

- C'est une démarche récente, elle doit encore faire ses preuves.
- Le besoin de flexibilité n'est peut-être pas encore la priorité de tous les concepteurs de système ou n'est pas identifié comme tel.
- La conception d'une structure à trois niveaux peut sembler complexe car elle exige dans sa représentation mentale un niveau supplémentaire.

Favoriser l'emploi d'une structure à trois niveaux est donc aussi un objectif dans notre validation de CAST. RAM3 a pour cela des atouts immédiats : il fournit le référentiel de modèles et de leurs instances, et ses capacités de prototypages permettent de démystifier la méta-modélisation vis-à-vis des concepteurs qui n'ont pas de compétences dans ce domaine (sans oublier que la définition du méta-modèle sera moins un casse-tête pour le concepteur grâce aux mécanismes réflexifs de RAM3). Toutes

les fonctionnalités présentées dans le cadre de M3CT ne feront que renforcer notre capacité à favoriser l'utilisation d'une structuration en trois niveaux.

3.2.2 Éditeur de liaison et prototypage

L'implémentation de M-CAST n'est pas terminée. Dans la partie 1.3.1, nous décrivions les éléments restants à implémenter. Pour une validation solide et donc une utilisation dans un contexte presque commercial, l'ergonomie de l'éditeur de liens d'adaptation aura besoin d'être améliorée. À cette fin, les compétences de notre laboratoire sur les IHM seront mises à profit. Deux éléments de réflexion peuvent débuter cette amélioration. Le premier est que les interfaces utilisateurs doivent fournir une assistance à la définition des liens. Elles sont mêmes tenues de favoriser la compréhension du fonctionnement de M-CAST (pour ne pas obliger le concepteur à assimiler complètement la documentation de M-CAST). La deuxième réflexion porte sur la présence d'une faculté d'observation constante du comportement des liens d'adaptations pendant la définition. Cela correspondrait par exemple à l'intégration dans l'éditeur, de liens, d'une instance/exemple (choisi par le concepteur) qui indiquerait, par exemple à droite de chaque caractéristique cible, le résultat de l'adaptation. Cette illustration concrétiserait pour le concepteur chaque étape de la définition des liens.

Le prototypage des services d'adaptation (AS) repose sur RAM3. Parce que l'implémentation de celui-ci est pratiquement terminée, la possibilité de prototyper les AS est presque déjà disponible. Il ne reste qu'à intégrer "la notion d'adaptateur" au sein de RAM3. L'association d'artefacts graphiques, présentée dans la partie précédente, améliorerait grandement l'ergonomie des éditeurs de modèles et donc probablement le prototypage des AS. Il reste à voir si la recherche d'erreurs ne devient pas plus limitée.

3.2.3 *CAST

Afin d'augmenter les chances d'application de CAST au plus grand nombre de systèmes possibles, une des perspectives de travail vis-à-vis de CAST est d'en proposer une version *CAST, c'est-à-dire multi-supports. Le premier aspect d'une telle version est l'application de CAST à d'autres supports. Le problème majeure est que seul le MOF propose 'quelque chose' pour les référentiels de modèles (en oubliant qu'il ne propose rien pour les instances de modèles). Il faut alors déduire des systèmes à trois niveaux existant la méthode de méta-modélisation (i.e. définition des concepts de types et d'instances) qu'ils utilisent (et qui constitue un équivalent du MOF). Le traitement de ce premier aspect permettra de vérifier si le formalisme d'adaptation de CAST peut lier des méta-modèles définis dans des méta-méta-modèles différents. Ce qui nous amène au deuxième aspect de *CAST : lier des méta-modèles définis dans des formalismes différents. Si les méta-modèles sont décrits dans des formalismes fortement différents (ex : objet ⇔relationnel), la définition des liaisons risquent d'être délicate voir infaisable. Ce deuxième aspect de *CAST demeure une question très ouverte.

3.2.4 CAST pour la création de fédération

Le principal problème de CAST dans une utilisation "à grande échelle" est la possibilité d'un grand nombre d'AS différents. S'il y a un grand nombres d'AS, c'est qu'il y a aussi un grand nombre de systèmes, et on peut considérer que le nombre d'informaticiens a même d'effectuer les liens d'adaptation est grand. Toutefois si cinq entreprises, avec chacun un système différent, coopèrent fortement ensemble (i.e. chacun coopère avec les quatre autres) et constituent ainsi une petite fédération, le nombre d'AS à créer est élevé : 10 !

La solution souvent utilisée dans les bases de données ou autres fédérations est de passer par un langage pivot. Avec cette approche, les concepteurs ne définissent qu'un seul AS pour chaque système : liens d'adaptation entre le méta-modèle du système et le méta-modèle pivot. Sur notre exemple, le nombre d'AS se réduit alors à 5. Nous avions expliqué qu'un langage pivot entraînait des pertes d'informations. Cependant, un manque de rigueur (du aux pertes d'informations) peut toujours être préféré à une charge trop conséquente de travail (causé par le nombre d'AS à définir). Pour cette raison, nous aimerions apporter à CAST la capacité de créer des fédérations. Le concepteur choisirait

(ou créerait) un méta-modèle pivot et pourrait ensuite rajouter des méta-modèles supplémentaires (correspondant chacun à un système) pour lesquels le concepteur définirait les liens adaptations uniquement avec le langage pivot. Cet ensemble *méta-modèle pivot + méta-modèles coopérants + liens d'adaptation vers le méta-modèle pivot* permettrait la génération d'un pseudo système pivot qui contiendrait les AS que l'on peut qualifier d'intermédiaires. L'adaptation d'un modèle d'un système A pour un système B se ferait en deux étapes : adaptation dans le méta-modèle pivot puis à partir du résultat adaptation dans le méta-modèle B (cf. figure 98).

La finalisation de CAST (déjà évoquée dans la partie 1) et son intégration dans un projet en partenariat avec des entreprises nous permettrait de valider de manière solide l'approche CAST (destiné essentiellement à un contexte industriel). Les améliorations présentées ici (ergonomie, création d'exportation MOF rapide) favoriseraient cette validation (*CAST et la création de fédération sont plus des ouvertures que des réelles perspectives de travail immédiat). Un avantage que nous possédons pour un tel projet est l'implémentation déjà fortement avancée de CAST (et RAM3).

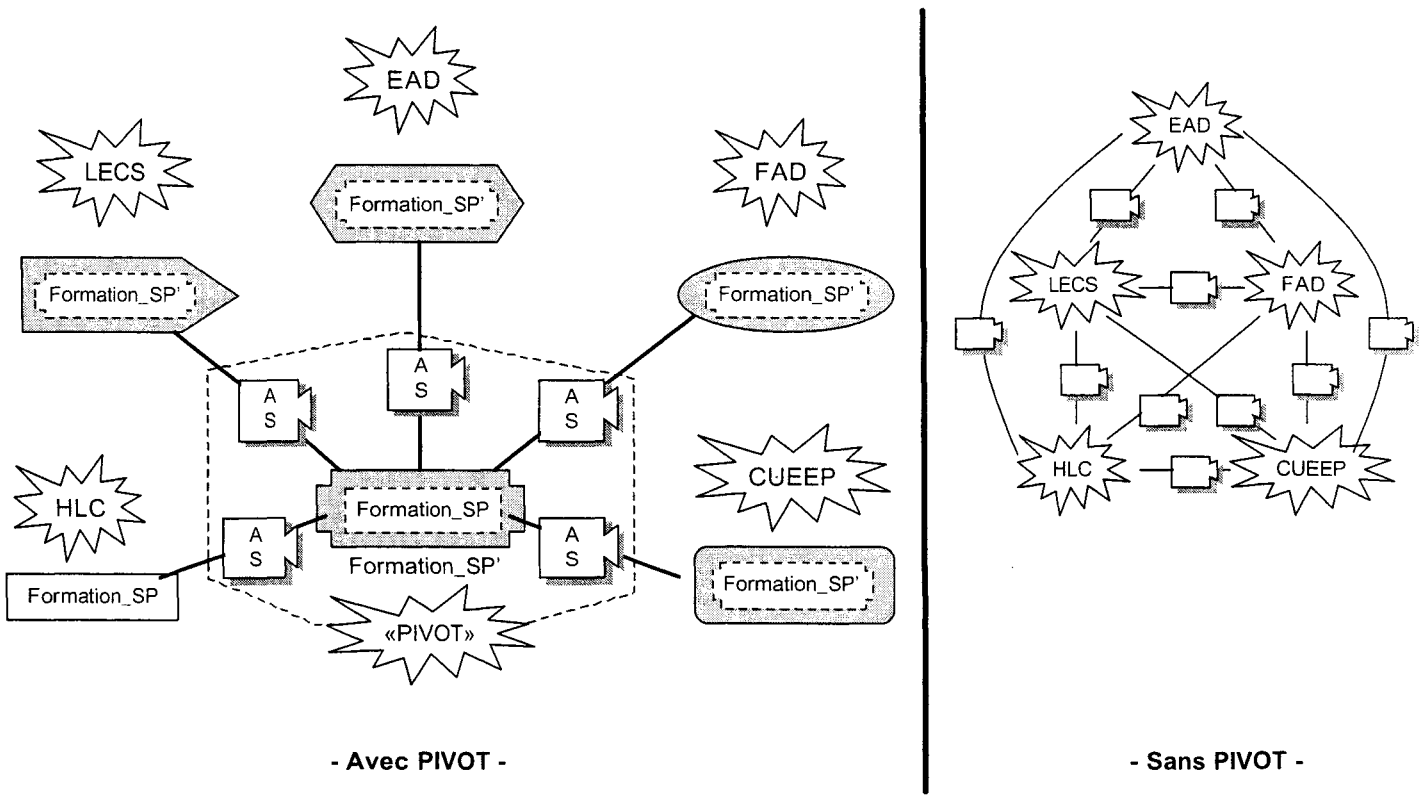


figure 98. Un système PIVOT pour la création d'une fédération

3.3 Le projet DARE

Le système DARE que nous avons présenté au chapitre 3 est en fait l'implémentation (partielle) d'une approche conceptuelle qui constitue le cœur du *projet DARE*. L'objectif de ce projet est d'offrir un support informatique à l'activité coopérative. La Théorie de l'Activité est au centre du projet et fournit la majeure partie des aspects conceptuels qui sont à la base de l'implémentation du système DARE résultant. M-CAST est d'ailleurs un complément à ce projet car il apporte au système DARE des capacités d'extensibilité supplémentaires en lui permettant de coopérer avec d'autres systèmes.

Le système DARE n'est qu'une première "tentative" de fournir un support informatique à la Théorie de l'Activité (cf. chapitre 3). Outre le système qui reste très prototypaire, le méta-modèle de DARE n'est pas complet : la notion de division du travail (relation objet-communauté), autre partie importante de la Théorie de l'Activité, n'y est pas complètement représentée. Une entière

représentation implique l'ajout à ce méta-modèle d'une dimension temporelle, de règles de coordination et de concepts pour la description d'organisation.

L'implémentation d'une nouvelle version du système DARE a débuté récemment et l'utilisation de RAM3 fait partie de ce nouveau projet. La première section indique l'apport de RAM3 dans la définition même du méta-modèle. Le gain de temps dans l'implémentation est l'objet de la seconde section. Les avantages de RAM3 (par rapport à SmallTalk employé pour la première version du système DARE) dans une perspective d'utilisation commerciale sont ensuite présentés. L'utilisation des méta-modèles additionnels et les apports de DARE pour le MOF constituent les deux dernières sections.

3.3.1 Prototypage du méta-modèle de DARE

La prochaine version du méta-modèle de DARE va intégrer d'autres notions de la Théorie de l'Activité. Le concept de Ressource fera son apparition. Les concepts de Tâche et d'Outil vont hériter de ce nouveau concept. La dimension temporelle va aussi apparaître avec les concepts de Transition et de Condition. Tous ces ajouts vont définir un nouveau méta-modèle. Cette opération n'est pas élémentaire. Nous l'avions expliqué dans le chapitre 6, la méta-modélisation (comprenant un niveau d'abstraction supplémentaire) est un exercice mental plus difficile que la modélisation. RAM3 a commencé à être utilisé après qu'une première nouvelle version du méta-modèle de DARE fut définit. Celle-ci fut testée avec RAM3 et le résultat correspond à notre argumentation du chapitre 6 vis-à-vis du prototypage : ce dernier est obligatoire. Le concept d'élément a du être défini (dont hérite tous les concepts), l'association d'héritage entre rôles s'est révélée problématique (elle sera donc peut-être supprimée) et un nouveau concept s'est révélé obligatoire, le concept d'Événement. Ce sont la création de modèles (issus de ce nouveau méta-modèle) et la création d'instances de ces modèles, opérations offertes par RAM3, qui ont permis de mettre en avant ces problèmes ou ces lacunes. Alors que pourtant sur papier tout paraissait cohérent. Sans utiliser RAM3 il faudrait implémenter le méta-modèle (sur M1 et M-zéro) et tester celui-ci ensuite. À chaque incohérence il faudrait revenir à la définition du méta-modèle, le modifier, le ré-implémenter, le ré-exécuter et recommencer ce cycle à la prochaine incohérence. C'est ce qu'évite RAM3.

Le prototypage ne se termine pas là. En modifiant les fonctions d'affichage de RAM3 pour les tâches, les rôles, les ressources, les transitions, ... pour les adapter aux utilisateurs finaux (car celles de RAM3 sont peut-être trop brutes), on peut aussi faire intervenir ces derniers et voir si les nouveaux concepts leur semblent compréhensibles et si cette compréhension rejoint celle du (ou des) concepteur(s) de DARE.

3.3.2 Moteur à trois niveaux et interfaces utilisateurs

Lorsque le méta-modèle de DARE sera défini de manière stable, l'implémentation du système pourra débuter. Celle-ci consiste à :

1. opérationnaliser le méta-modèle - c'est-à-dire à créer un référentiel à deux niveaux (modèles et instances) basé sur le méta-modèle -,
2. rendre accessible ce référentiel de l'extérieur,
3. implémenter le comportement des concepts de type et d'instance (ex : la gestion du partage des outils qui sont des applets Java, l'exécution d'une action qui invoque des méthodes Java de l'applet, ...),
4. à fournir un service de persistance,
5. à concevoir les interfaces utilisateurs clientes : la modélisation des activités, la connexion d'un utilisateur, l'apparition du statut de l'activité et des composants/outils,
6. implémenter les aspects dépendants de la plate-forme utilisé, c'est-à-dire un environnement Web (page HTML, navigateur et serveur Web, applet, ...).

L'apport de RAM3 est principalement située au niveau de l'opérationnalisation du méta-modèle : il peut être utilisé tel quel au sein de tout programme Java et donc permettre à ce dernier de charger un méta-modèle et d'instancier (sur deux niveaux) celui-ci (1). Le référentiel est compatible Corba et donc accessible à distance (2). Le service de persistance peut être celui de RAM3 (4). Les interfaces utilisateurs peuvent être prototypées avec RAM3 afin de les valider avec les utilisateurs (5). L'utilisation de l'API RAM3 (get, set,...) devra ensuite être traduite en invocations Corba pour les interfaces utilisateurs finales.

3.3.3 Utilisation industrielle

L'implémentation Smalltalk (du référentiel) du système DARE actuel risque de rendre très sceptique une société qui souhaiterait commercialiser le système DARE ou une spécialisation de celui-ci. Lorsque la société Archimed [ARC 02] eu l'intention de commercialiser le Campus Virtuel [ACV 02] (développé au sein de notre laboratoire), elle fut perplexe quant à l'utilisation de Smalltalk (pour l'implémentation d'ODESCA à la base du Campus Virtuel) :

- Il lui fallait acquérir la licence de commercialisation car Archimed développait ses produits avec d'autres outils.
- Des développeurs aurait du acquérir de très bonnes compétences en Smalltalk pour la maintenance du Campus.
- La persistance et donc la restauration de données en cas de panne ne sont pas gérées parfaitement par Smalltalk.

DARE est en partie implémenté en Smalltalk car il utilise les mécanismes réflexifs que ce langage propose (ex : création de classes dynamiques). L'utilisation de RAM3 permettra de supprimer cette partie Smalltalk et d'avoir une implémentation intégralement en Java. Cette nouvelle configuration provoquera sûrement moins de réticences pour un futur partenaire industriel.

Le moteur de RAM3 peut paraître lent sur de vieilles machines (Pentium 3 – 500MHz – 128 Mo RAM). Néanmoins, pour DARE, le moteur prendra place sur un serveur qui est normalement une machine puissante. La lenteur devrait donc disparaître. De plus, le langage Java devrait intégrer prochaines des mécanismes d'intercession. Nous ne manquerons de remplacer le plus possible les mécanismes d'intercession de RAM3 au profit de ceux de Java afin de gagner en rapidité d'exécution.

Le bus Corba qu'utilise le système DARE est VisiBroker. Orbacus, utilisé par RAM3, est réputé pour avoir un grand respect de la norme Corba, et offre a priori une meilleure ouverture que Visibroker. De plus, d'après notre expérience, Orbacus semble plus stable que celui-ci.

3.3.4 DARE et le MOF

Avec le système DARE, la modélisation intervient dans un contexte coopératif : les participants modifient le modèle de leur activité dans une activité elle-même coopérative. Dans cette "méta-activité", subsistent les droits d'accès aux ressources, qui sont ici les éléments du modèle. Cet aspect 'restriction' peut aussi apparaître dans un système de workflow où les utilisateurs peuvent changer eux-mêmes leurs modèles de processus. Dans ce contexte, est mise en évidence la nécessité de définir des droits d'accès. L'aspect coopératif (surtout dans des supports distribués) pourrait donc apparaître dans le MOF : les règles de génération fourniraient des mécanismes de restriction d'accès. Ceci implique que dans la définition même du méta-modèle, il serait possible d'indiquer quels sont les éléments susceptibles de se voir adjoindre des droits d'accès (et sur lesquels cela est inutile).

C'est une perspective d'étude. Une question pourrait se poser alors : en arrivera-t-on à des profiles MOF pour gérer toutes ces extensions (M-zéro, aspects coopératifs) ? Notre réponse est non.

1. Les extensions M-zéro que nous avons apporté au MOF étaient, pour nous, obligatoires. La description d'un méta-modèle n'est pas complète sans expliciter le comportement des instances de ses instances. Ce n'est pas un rajout, un aspect supplémentaire intéressant à intégrer, mais c'est essentiel si on veut rendre opérationnel le méta-modèle.

2. L'apparition d'aspects coopératifs serait plutôt à avoir comme une nouvelle remise en cause du paradigme objet utilisé par le MOF. Le concept de Rôle est de plus en plus cité au sein des travaux sur la pratique de la modélisation objet. Pour Kristensen [KRI 97], nous pensons et nous exprimons en termes de rôles. Il paraît alors plus logique de modéliser à l'aide de rôles. Dans le même ordre d'idées, Halpin [HAL 01] préconise une approche par les faits et évoque la verbalisation par les faits et les règles. D'un point de vue plus technique, Wegmann [WEG 00] explique que les rôles apportent une meilleure description du *comportement collaboratif*. Cette idée est partagée par Reenskaug [REE 96] pour qui la notion de Rôle permet de distinguer les responsabilités et les capacités d'un objet dans une collaboration. Selon Kristensen, la visibilité, les dépendances et la dynamicité sont des propriétés qu'il est plus facile d'exprimer à l'aide de rôles. De telles considérations ont été prises en compte par l'OMG. UML intègre, depuis la version 1.3, un nouveau type de diagramme : les diagrammes de collaboration de classes. Il s'agit d'une collaboration de *ClassifierRoles*, c'est-à-dire des spécifications de caractéristiques structurelles et comportementales que des classes peuvent respecter. Le concept ClassifierRole étend le concept d'interface (limité aux caractéristiques comportementales). L'apparition de ce type de diagramme au sein d'UML confirme l'intérêt du concept de rôle dans la modélisation objet.

3.3.5 Les méta-modèles additionnels

Un des objectifs du système DARE est de proposer une solution au problème ontologique rencontré par ODESCA (cf. chapitre 1), c'est-à-dire le problème de divergence sur le sens des concepts pour le même méta-modèle selon que l'on soit utilisateur ou concepteur. Avec le méta-modèle issu de l'AT, G. Bourguin espère supprimer cette hétérogénéité sémantique.

L'utilisation de méta-modèles additionnels, présentés dans le cadre de M3CT, offrirait de nouvelles perspectives pour solutionner le problème ontologique. Cette fonctionnalité permettrait au concepteur du système DARE d'instancier les notions de l'AT dans différents méta-modèles qui représenteraient différents types de participants : utilisateur, spécialiste, informaticien, ... Il n'y aurait plus un outil de modélisation mais plusieurs, chacun avec un méta-modèle différent. Le système DARE fournirait à l'utilisateur pour la modélisation des activités un outil adapté à ses compétences. Le fonctionnement des méta-modèles additionnels étant basé sur des adaptateurs autour d'un référentiel unique (i.e. partage de ce référentiel), il convient parfaitement à la modélisation coopérative de DARE.

Ces préoccupations vis-à-vis de l'adaptation de l'outil de modélisation (des activités) à l'utilisateur peuvent sembler être un simple problème du domaine des IHM. Et il peut sembler suffisant de penser qu'il s'agit simplement de concevoir des interfaces qui fournissent plusieurs vues du méta-modèle de DARE. Ceci est en partie vrai. Mais dans notre approche (par les méta-modèles additionnels), la conception de ces outils de vues supplémentaires est plus formelle. Elle est donc pas conséquent plus explicite que du code, c'est-à-dire plus facilement compréhensible, évolutive, applicables à d'autre plates-formes/environnement, ...

3.4 Mes priorités

La validation de CAST peut sembler la poursuite logique de mes travaux de recherches. La validation consiste à terminer l'implémentation des différents modules de M-CAST puis à faire coopérer deux méta-groupware MOF à l'aide d'un service d'adaptation. Il serait inutile de se dépêcher de terminer l'implémentation de M-CAST vu l'absence actuelle de systèmes MOF. Pour cette raison, le développement a de fortes chances d'être effectué à travers des projets d'étudiants de DESS informatique.

Le développement de M3CT ne s'inscrit pas dans la thématique du laboratoire Trigone. Pourtant, dans la description des perspectives de recherche en rapport avec DARE, il apparaît très clairement qu'un tel outil serait d'une aide très précieuse dans le développement de la nouvelle version du système DARE. M3CT offrirait même des perspectives de fonctionnalités non envisageables autrement (à cause d'un développement trop lourd). Pour profiter de cet apport, notre idée est donc de

développer cet outil en second plan, que ce soit à travers un projet en partenariat avec une société spécialisée dans la conception d'AGL ou au travers de stages effectués au sein du laboratoire.

La poursuite immédiate de mes recherches va plutôt se porter sur le projet DARE. Ce dernier est, de nature, plus spécifique à la thématique du laboratoire Trigone.

Vis-à-vis de DARE, il s'agit de fournir les preuves, en terme de réalisation, des apports possibles de RAM3 (c'est-à-dire un outil de prototypage et d'opérationnalisation de méta-modèles) au domaine du TCAO/ACAO. C'est un travail, certes, assez axé sur l'implémentation, mais il a pour second avantage de proposer un premier système MOF, c'est-à-dire un élément essentiel à la validation de CAST.

La figure 99 résume mes priorités.

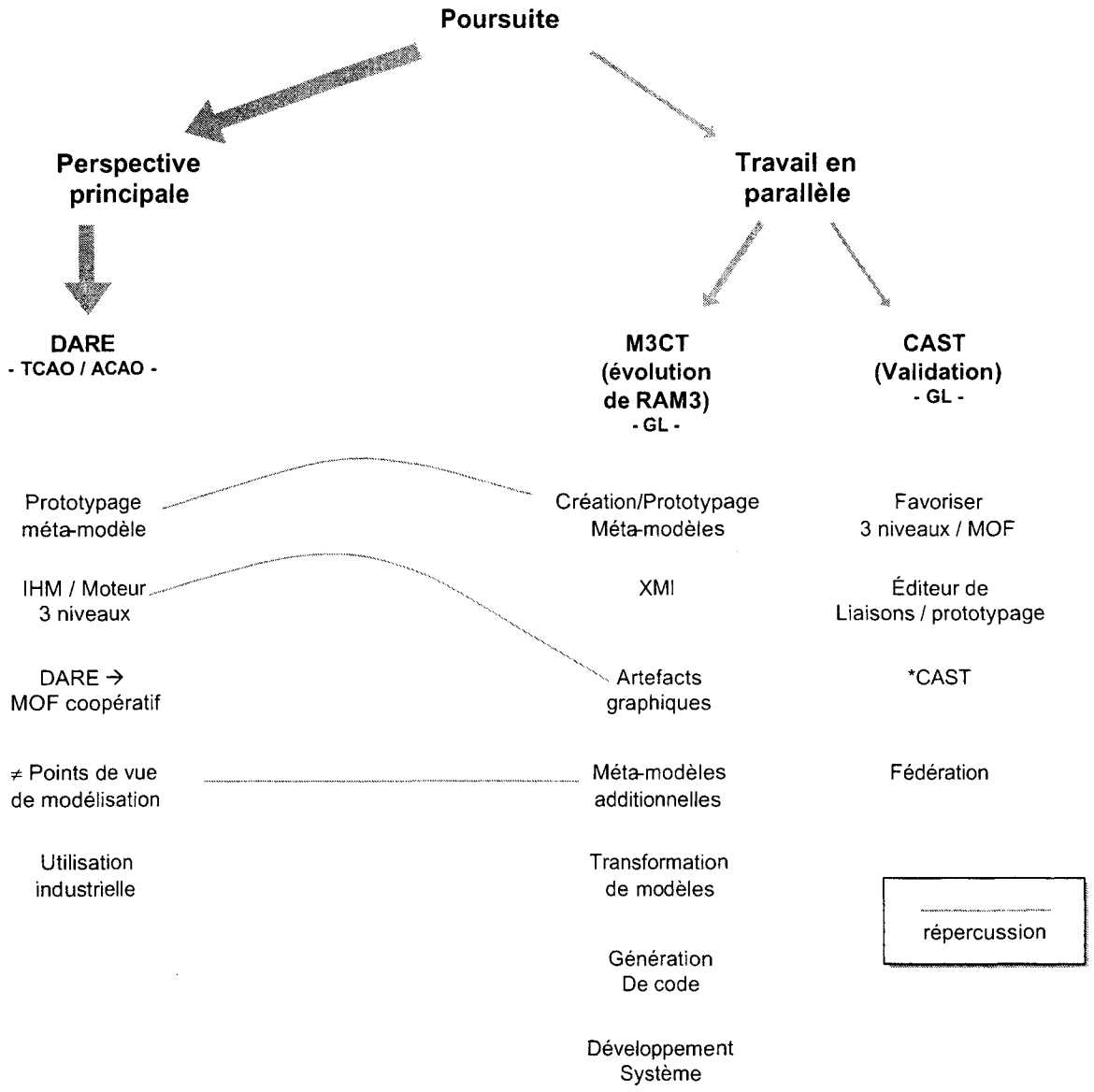


figure 99. Perspectives de Recherche et Priorités

Conclusion

Nous avons présenté dans le chapitre 7 le bilan de nos travaux et les perspectives éventuelles pour nos recherches futures. Dans cette conclusion nous aimerions toutefois revenir sur la démarche que nous avons adoptée au cours de cette thèse.

Aujourd'hui, un système informatique dune entreprise doit faciliter (ou tout au moins ne pas freiner) la réactivité de celle-ci et les éventuelles coopérations avec d'autres entreprise. Ces deux nécessités et leur répercussion ont fait naître un nouveau problème : la coopération de systèmes flexibles. Les solutions actuelles d'interopérabilité comme l'utilisation de standard(s), la médiation externe ou l'interaction par spécifications ne sont pas adaptées au caractère dynamique de ces systèmes et au type d'implication des utilisateurs qui y a lieu. C'est par un ensemble de contraintes (accessibilité, visibilité et flexibilité des liens de coopération et un champ d'action plus faible) que nous avons identifié que nous avons pu démontré cette inadaptation.

Notre étude, visant à proposer une solution à la coopération inter-organisationnelle, s'est alors focalisée sur les groupware, car ils sont les plus à même de gérer les activités au sein d'une organisation. Notre problématique portant sur les systèmes flexibles, nous avons choisi de nous pencher sur les méta-groupware : des groupware adoptant l'approche par méta-modèles (où les modèles de coordination et de collaboration sont modifiables dynamiquement par les utilisateurs). Le nombre de méta-modèles utilisés par les méta-groupware est pratiquement égal au nombre de ces derniers. Les bases de ces méta-modèles peuvent même être fortement différentes. Pour cette raison, notre étude s'est concentré sur le seul dénominateur commun à ces méta-groupware : l'approche par méta-modèles. Si nous avions inclus des concepts propres au workflow ou à des approches orientées collaboration, nous aurions limité la porte de notre étude, la rendant incompatible avec des méta-groupware différents.

Le résultat de notre étude est une solution à la coopération inter-organisationnelle adaptée à l'approche par méta-modèles. Le principe de cette solution est de placer des services d'adaptation qui permettent aux utilisateurs, à partir de leur système, de voir et de manipuler des modèles externes et de les associer avec leurs "propres" modèles (association bilatérale). Les modèles ainsi liés forment une organisation virtuelle union des organisations coopérantes. Il ne s'agit pas d'association de modèles transformés mais d'une réelle liaison entre modèles "vivants" dans des systèmes différents. De cette manière, les instances/réalisations de ces modèles se synchronisent entre elles automatiquement et la coopération entre organisations a donc lieu. La solution accompagnant le principe précité est CAST – Creation Adaptation Service Tool. Il a pour objectif d'accélérer le développement de tels services grâce à un formalisme graphique de liaison, un modèle générique de services d'adaptation, et une prise en compte des relations entre modèles et leurs instances (qui évite certaines définitions). CAST reste conceptuel, c'est-à-dire sans dépendance vis-à-vis d'une plateforme.

Pour valider notre approche, il nous fallait concrétiser celle-ci en l'implémentant. Les modèles ayant une place primordiale dans notre approche, l'élément clé de cette réalisation devait être un bus de dialogue pour les modèles. Pour cela, nous avons choisi le MOF – Meta-Object Facility. Il s'agit du seul standard de représentation de modèles de tout types. Nous avons du effectué un travail important pour adapter le MOF à l'approche par méta-modèles qui fait intervenir des liens entre modèles et leurs instances (liens inexistant dans le MOF). Pour implémenter M-CAST (CAST appliqué au MOF), nous avons aussi du développer un environnement dédié au MOF. La raison de ce développement provient de la nécessité de prototyper les services d'adaptation avant leur utilisation. Ces services n'étant que objets MOF "adaptateurs", prototyper ces services revient à créer de des objets MOF et à les éditer. Concernant cet aspect, l'outillage MOF est peu fourni. D'où le développement de notre propre outil MOF : RAM3 – RApid Manipulation of Mof Metadata. Les caractéristiques réflexives puissantes de cet outil lui permettent aussi d'être aussi un environnement de prototypage de méta-systèmes compatibles MOF.

Mis à part l'implémentation de M-CAST qui reste à terminer, les enjeux de départ ont été atteints. Nous avons d'abord identifié les objectifs à atteindre (accessibilité, visibilité, flexibilité...) pour toute solution de coopération inter-organisationnelle où les systèmes informatiques sont flexibles. Nous avons étudié les solutions d'interopérabilité actuelles et montré leur faiblesse vis-à-vis du précédent problème. Nous avons décrit de manière précise les mécanismes à la base de l'approche la plus efficace pour construire un système flexible. Cette description nous a permis de définir une solution complètement adaptée à la coopération de systèmes flexibles.

Nos travaux ont aussi des apports annexes à notre problématique de départ. L'apparition d'un bus d'échange de modèles (le MOF) a rencontré beaucoup d'incompréhension. Nous avons ici clarifié l'utilité d'un tel bus. L'ensemble des travaux présentés dans ce manuscrit constitue même un cas important d'utilisation. CAST apporte un nouveau type de transformation de modèles, l'adaptation, ce qui vient augmenter l'outillage de la transformation dans le contexte de l'interopérabilité. Enfin la description précise des liens entre modèles et leurs instances sera très utile dans les futures plateformes de développement orientées modèles, tel que le MDA – Model Driven Architecture de l'OMG.

Références

[ABO 94] Abbot, K. R. and Sarin, S. K., *Experiences with Workflow Management: Issues for The Next Generation*, Computer-Supported Cooperative Work (CSCW 94), 1994.

[ACV 02] *Archimed – Campus Virtuel 2.2*, http://www.archimed.fr/ABV/framesetPortail.asp

[ALT 00] Alter S., *Same words, different meanings: are basic IS/IT concepts our self-imposed tower of babel?*, FUNDAMENTALS OF IS, Communications of the Association for Information Systems, April 2000, Volume 3, Article 10.

[ARC 02] *Archimed Home page*, www.archimed.fr

[AUT 02] *Autodesk - Building Design Solutions - Solutions de workflow*, http://www.autodesk.fr

[BEZ 94] Bézivin Jean, Lanneluc Jérôme, Lemesle Richard (1994). *Un réseau sémantique au coeur d'un AGL.* Conférence LMO'94, Grenoble, , pp. 13-24

[BEZ 98] Bézivin, Jean and Lemesle, Richard. "The sBrowser: a Prototype Meta-Browser for Model Engineering." Proceedings of OOPSLA 98, Vancouver, Canada, 18-22 Oct 1998

[BIZ 02] Microsoft BizTalk Sever, http://www.microsoft.com/biztalk/default.asp

[BLA 00] Blanc, X.; Gervais, M.-P.; Le Delliou, J., *The Specifications Exchange Service of an RM-ODP Framework*, EDOC 2000, 25-28 September, Makuhari, Japan, proceedings.

[BLY 97] Blyth A., *Business Process Re-Engineering : Challenges for the Future*, SIGGROUP Bulletin, Vol. 18 No. 3 (December 1997).

[BOB 93] Bobrow D.G., Gabriel R.P., White J.L., *CLOS in context*, chapitre 2 du livre Object-oriented programming : the CLOS perspective, édité par Andreas Paepcke, 1993.

[BON 98] Charles Bontempo, George Zagelow, *The IBM data warehouse architecture*, Communications of the ACM, Volume 41 , Issue 9 (September 1998), pp 38-48

[BOR 00] UML Profile, Born, M., Holz, M. and Kath, M., *A Method for the Design and Development of Distributed Applications using UML*, International Conference on Technology of of Object-Oriented Languages and Systems (TOOLS Pacific), Sydney Australia, November, 2000

[BOU 00] Bourguin G., *Un support informatique à l'activité coopérative fondé sur la Théorie de l'Activité : le projet DARE*, Th. De Doctorat en Informatique, Université des Sciences et Technologies de Lille, Juillet 2000, n° 2753.

[BOU 00b] Jean-Paul Bouchet et Sébastien Boucard, *Utilisation de méta-modèles standardisés autour du MOF pour l'ingénierie de systèmes*, 3ème réunion META, 17 Mai 2000, http://www.lip6.fr/meta

[BOU 01] Bourguin G., Derycke A., Tarby J.C., *Au delà des Interfaces, la Co-évolution au sein des Systèmes Interactifs: Une Proposition fondée sur la Théorie de l'Activité*, à paraitre en version anglaise dans les actes de la conférence IHM-HCI 2001, 11 p.

[BOU 01b] Bourguin G., Le Pallec X., *Supporting Human Activities, The Meta-Level Issue*, actes de la conférence ICEIS 2001, July (7-10) in Setúbal/Portugal, 11 p.

[BOU 99] Bourguin G., *D.A.R.E. (Distributed Activities in a Reflexive Environment)*, A full day workshop at ECSCW'99 : Evolving use of groupware, 6th European Conference on Computer Supported Cooperative Work, Copenhagen, DENMARK, 12-16 September 1999, http://wwwold.telin.nl/events/ecscw99evo/participants.html, 2 p.

[BOX 00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, *Simple Object Access Protocol (SOAP) 1.1*, The World Wide Web Consortium 2000.

[BRE 02] Breton Erwan & Bézivin Jean, *Weaving Definition and Execution Aspects of Process Meta-Models*, in proceedings of the 35th Hawaii International Conference on System Sciences (HICSS-35) - Minitrack Software Technology/Integrated Modeling of Distributed Systems and Workflow Applications, Waikoloa, Hawaii, January 2002.

[BUS 98] Bussler, Chr., *Towards Workflow Type Inheritance*. In: Proceedings of the 1998 OOPSLA Workshop on the Implementation and Application of Object Oriented Workflow Management Systems. Vancouver 1998

[CAN 02] Canalda P., Godard C., *Coopetitive multi-enterprise process modelling: principles and guidelines on report*, MSCEAI' 02, Annaba – Algérie, 6-8 Mai 2002.

[CAR 02] Caron O., Geib J-M., Renaud E., *Vers de véritables composants EJB réutilisables*, Workshop OCM 2002, Nantes Juin 2002.

[CAR 97] Carlsen, S., Krogstie, J., Slvberg, A. and Lindland, O. I. *Evaluating Flexible Workflow Systems*, Hawaii International Conference on System Sciences (HICSS-30), Maui, Hawaii, 1997

[CHR 01] Christensen, Erik, et al., *Web Services Description Language (WSDL) 1.1.*, W3C Note. Mar. 2001. http://www.w3.org/TR/wsdl

[COL 02] Plateforme générique de travail Coopératif, INFORSID 02 – Forum Jeunes Chercheurs, Nantes Juin 2002, actes pp 419-420

[CWM 01] CWM Partners, "Common Warehouse Metamodel", OMG ad/2001-02-{01,02,03}

[DEM 86] Deming, W. E. 1986. *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering Study.

[DEM 97] De Michelis, G., Dubois, E., Jarke, M., Matthes, F., Mylopoulos, J., Papazoglou, P., Pohl, K., Schmidt, J.W., Woo, C., and Yu, E. *Cooperative information systems: a manifesto*. In Cooperative Information Systems: Trends & Directions, M. Papazoglou and G. Schlageter, Eds., Academic-Press, 1998, 315 - 363.

[DER 93] Derycke A. Kaye A., Participative modeling and design of collaborative learning tools in the CO-LEARN project. In G. Davis, B. Samways (eds), IFIP, Teleteaching 95 Conference, Trondheim, August 20-25,North-Holland, Amsterdam, pp. 191-200.

[DER 94] Derycke A. Viéville C., Real-time multimedia conferencing system and collborative learning. Collaboration Dialogue Technologies in distance education, Verdejo, F., Ceri, S. (eds), NATO ASI Series, Springer Verlag, Berlin, 1994, pp. 236-256.

[DIO 02] Dion Almaer, *Creating Web Services with Apache Axis,* http://www.onjava.com/lpt/a/onjava/2002/06/05/axis.html, Published on The O'Reilly Network (http://www.oreillynet.com/), 05/22/2002

[DMO 02] *dMOF 1.1*, http://www.dstc.edu.au/Products/CORBA/MOF/

[DOG 98] Asuman Dogac , Cevdet Dengi , M. Tamer Öszu, *Distributed object computing platforms*, Communications of the ACM September 1998, Volume 41 Issue 9

[DOM 02] *Honeywell DOME (Domain Modeling Environment) Homepage*, http://www.htc.honeywell.com/dome/

[DOV 97] Rick Dove, *Introducing Principles for Agile Systems*, Paradigm Shift International, http://www.parshift.com, (originally published 8/95 in Production Magazine, Gardner Publications - Revised 9/97).

[DST 02] *MOF Student Project Information - Generating MOF-based GUI Browsers*, http://www.dstc.edu.au/Research/Projects/MOF/Student-Info.html#browsers

[EDO 01] EDOC Partners, *Enterprise Distributed Object Computing*, 2001, OMG ad/01-06-09 and ad/01-08-18.

[EDU 02] *Report on Edubox*, http://www.unisa.ac.za/dept/buo/edubox-report/

[EIA 02] *The Electronic Industrial Alliance Homepage*, http://www.eia.org

[ENG 87] Engeström Y., *Learning by expanding: an activity-theoretical approach to developmental research*, Orienta-Konsultit Oy, Helsinki, 1987

[ERN 97] Ernst J., *Introduction to CDIF*, September 1997, http://www.eigroup.org/cdif/intro.html

[FER 89] Ferber, J., *Computational Reflection in Class based Object Oriented Languages.* In *OOPSLA '89 Proceedings.* 1989, pp. 317-326.

[FIN 02] *FINEOS Solutions Insurance, Workflow Manager,* http://www.fineos.com/solutions/bank_bs/workflow_manag.htm

[FLA 02] David Flater, *Impact of Model-Driven Standards*, Proceedings of the 35th Hawaii International Conference on System Sciences – 2002

[FYA 96] Fyad M. Cline M.P., *Aspects of Software Adaptability*, communications of the ACM, October 1996/Vol. 39, No. 10

[GEI 97] Geida A.S. *Foundations of the transformation approach to modeling and programming for integrated automation of technological processes planning*, International conference on Informatics and Control. ICI&C'97. June 9-13, 1997. S.-Pb.: SPIIRAS, 1997.- ? . 17-25.

[GEI 97b] Geib J., Gransart C., Merle P., *CORBA : Des concepts à la pratique*, Edition InterEditions, 1997.

[GEN 00] Guy Genilloud, Alain Wegmann, *On Types, Instances, and Classes in UML*, EPFL-ICA 2000

[GOD 99] Claude Godart, Olivier Perrin, and Hala Skaf. COO : *a Workflow Operator to Improve Cooperation Modeling in Virtual Processes.* In RIDE-VE'99, Sydney, Australia, 1999

[GOE 00] Goedicke M., Neumann G., Zdun U., *Object System Layer*, EuroPLoP 2000, Fifth European Conference on Pattern Languages of Programs 5 - 9 July 2000, Irsee, Germany

[GoF 95] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley, 1995

[GOL 89] Adele Goldberg and David Robson. *Smalltalk: The Language.* Addison-Wesley, 1989

[GRØ 94] Grøbæk K., Malhotra J., *Building Tailorable Hypermedia Systems: the embedded-interpreter approach Computer*, OOPSLA 94, Portland, Oregon USA.

[GRU 94] J. Grudin. *Groupware and social dynamics: Eight challenges for developers.* Communications of the ACM, 37(1):92-105, January 1994.

[HAL 01] Terry A. Halpin, 2001. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design.* (Morgan Kaufmann Series in Data Management Systems) Morgan Kaufmann Publishers.

[HAN 98] Y. Han, A. Sheth and C. Bussler, *A Taxonomy of Adaptive Workflow Management,* Workshop of the 1998 ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, USA, November 1998

[HAR 97] Hardwick M., Richard Bolton, *The industrial virtual enterprise*, Communications of the ACM September 1997, Volume 40 Issue 9

[HEI 95] Heiler S., *Semantic Interoperability*. ACM Computing Surveys 27:2, 1995, pp 271-73.

[HO 99] UMLaut, Wai Ming Ho, Jean-Marc Jezequel, Alain Le Guennec, and Franois Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999

[HOE 01] Hoogstoel F., Kolski, Christophe (sous la direction de), *Environnements évolués et évaluation de l'IHM – Interaction homme-machine pour les SI 2,* Paris, Hermès Science Publications, 2001.

[HOO 95] Hoogstoel F., *Une approche organisationnelle du Travail Coopératif Assisté par Ordinateur. Application au projet Co-Learn*, Th. De Doctorat en Informatique, Université des Sciences et Technologies de Lille, 1995, n° 1487.

[HSP 02] *Health Solution Plus : Workflow Management, Claims Processing,* http://www.soltech.com/

[IDL 99] *IdlScript , OOC and LIFL. CORBA Scripting* - Joint Revised Submission. OMG TC Document orbos/99-07-17. Object Management Group, August 1999.

[IEE 90] *IEEE standard computer dictionary : compilation of IEEE standard computer glossaries*, 610 / sponsor, Standards Coordinating Committee of the IEEE Computer Society, New York : Institute of Electrical and Electronics Engineers, c1990

[INT 02] *The Interoperability Focus home page*, http://www.ukoln.ac.uk/interop-focus/

[ION 02] *IONA - Solutions - Corba Products*, http://www.iona.com/products/orbacus_home.htm

[JOH 98] Johnson R., *Dynamic Object Model Architecture*, non publié, http://st-www.cs.uiuc.edu/users/johnson/papers/dom/

[JON 98] Katherine Jones, *An introduction to data warehousing: what are the implications for the network?*, International Journal of Network Management, Volume 8, Issue 1, January–February 1998, pp. 42-56

[JSR 02] *JSR-000040 JavaTM Metadata Interface Specification*, http://www.jcp.org/aboutJava/communityprocess/review/jsr040/

[KAM 00] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman, *Techniques for supporting dynamic and adaptive workflow*, Computer Supported Cooperative Work, vol. 9, no. 3/4, August, 2000

[KAM 98] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi. *Supporting Distributed Workflow using HTTP.* Proceeding of the 5 th International Conference in Process, May 1998

[KAN 88] H. Kanakia , D. Cheriton, *The VMP network adapter board (NAB): high-performance network communication for multiprocessors*, ACM SIGCOMM Computer Communication Review , Symposium proceedings on Communications architectures and protocols August 1988, Volume 18 Issue 4

[KAR 01] Karacapilidis N., *On the Development of an E-Business Oriented Workflow Management System*, In Proceedings of the BITWorld 2001 International Conference on Business Information Technology, Cairo, Egypt, June 4-6, 2001

[KIC 91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, chapter 5,6. MIT Press, 1991.

[KLE 96] Klenke K., *IS leadership in virtual organizations*, Proceedings of the 1996 conference on ACM SIGCPR/SIGMIS conference April 1996

[KOC 00] Kock N., *Benefits for virtual organizations from distributed groups*, Communications of the ACM November 2000, Volume 43 Issue 11

[KOP 02] Koper Rob , *Pedagogical meta-model behind EML*, Document prepared for the IMS Learning Design Group; 'Modeling Units of Study from a Pedagogical Perspective: the pedagogical meta-model behind EML

[KRI 97] Kristensen B., *Subject Composition by Roles*, OOIS'97 – 1997 international conference on object oriented information systems proceedings, Brisbane, 10-12 november 1997

[KRU 97] Bobby Krupczak , Kenneth L. Calvert , Mostafa H. Ammar, *Increasing the portability and re-usability of protocol code*, IEEE/ACM Transactions on Networking (TON) August 1997, Volume 5 Issue 4

[KUT 91] Kuutti K., *The concept of activity as a basic unit of analysis for CSCW research*, Proceeding of the second ECSCW'91 conference, Kluwers Academics Publishers, 1991, pp 249-264

[LAV 96] R. Greg Lavender , Douglas C. Schmidt, *Active object: an object behavioral pattern for concurrent programming*, Pattern languages of program design 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1996

[LEI 97] Y. Lei and M.P. Singh, *A Comparison of Workflow Metamodels*, ER'97 Workshop 4 Proceedings, 1997

[LEM 00] Semantor, Lemesle R., *Techniques de Modélisation et de Méta-modélisation*, Th. De Doctorat en Informatique, Université de Nantes, 2000.

[LEM 98] R. Lemesle, *Transformation Rules Based on Meta-Modeling*. EDOC'98, San Diego, [November 1998].

[LEP 00] Le Pallec X., Bourguin G., Peter Y., *Gestion de méta-données avec le Meta Object Facility*, actes de la conférence OCM'2000, Objets, Composants, Modèles, passé, présent, futur, Nantes, FRANCE, 18 mai 2000, pp. 101-112

[LEP 01] Le Pallec X., Vantroys T., *A Cooperative Workflow Management System with the Meta-Object Facility*, EDOC 2001, IEEE, Seattle, Washington USA 4-7 Septembre, Proceedings pp 273-281

[LEP 01b] Le Pallec X., Bourguin G., *RAM3 : un outil dynamique pour le Meta-Object Facility*, actes de la conférence LMO'01, Langages et modèles à objet, Le Croisic, FRANCE, 29-31 Janvier 2001, revue L'Objet, Logiciel, base de données, réseaux, Hermes, vol. 7 - n°1-2/2001, pp 79-94.

[LES 98] Lesueur B., Revault N., Sunyé G., Ziane M., Blain G., *Using the Méta-Gen modelling and development environment in the FIBOF Esprit Project*, in ECOOP-98 Automating the object-oriented software development workshop, Bruxelles, 1998

[LEY 01] F. Leyman., *Web Services Flow Language (WSFL)*. Technical report, IBM, May 2001

[M3J 02] *M3J Project*, http://www-src.lip6.fr/meta/Projets/M3J/

[MAE 87] Pattie Maes. *Concepts and experiments in computation reflection*. OOPSLA'87, Sigplan Notices, Vol. 22 N'12, Decembre 1987.

[MAN 93] Tomi Männistö, Hannu Peltonen, Kari Alho, and Reijo Sulonen. *A framework for long term information management of product configurations*. Technical Report TKO-B105, Helsinki University of Technology, Laboratory of Information Processing Science, 1993

[MAN 98] Manolescu D., Johnson R., *Dynamic Object Model and Adaptive Workflow*

[MAN 98b] Dragos,-Anton Manolescu and Ralph E. Johnson, *Patterns of Workflow Management Facility*, non publié, http://www.uiuc.edu/ph/www/manolesc/Workflow/PWFMF/.

[MAN 99] Peter Manhart, *A system architecture for the extension of structured information spaces by coordinated CSCW services*, Proceedings of the international ACM SIGGROUP conference on Supporting group work November 1999

[MDA 02] Object Management Group, *Model Driven Architecture: The Architecture Of Choice for a Changing World*, 2001, www.omg.org/mda/

[MDR 02] *Metadata Repository (MDR) home*, http://mdr.netbeans.org/

[MED 95] C.B. Medeiros, G. Vossen, and M. Weske. *WASA: A workflow-based architecture to support scientific database applications*. In International Workshop and Conference on Database and Expert Systems Applications (DEXA), London, U.K., Sept 4--8 1995

[MEG 02] MEGA, *Business Process Modeling & IT Mapping Solutions*, http://www.mega.fr

[MER 96] CorbaWeb, Philippe Merle, Christophe Gransart, and Jean-Marc Geib. *CorbaWeb: A WWW and Corba Worlds Integration*. In The 2th COOTS, Workshop on Distributed Object Computing on the Internet, Toronto, Canada, June 1996.

[MET 96] *What is a Meta CASE ?*, http://www.cs.ualberta.ca/~softeng/Metaview/meta_case.html

[MIC 99] marshalling M. Michel, A. Schaff, and J. E. Devaney. *Managing data-types : the corba approach and automap/autolink, an mpi solution*. Proceedings of the third MPI Developer's and User's Conference, page 143, 1999. Presented at MPIDC'99.

[MOO 00] Connie Moore, *Workflow Goes Mainstream*, Giga Information Group, April 20, 2000, http://www.gigaweb.com

[MOR 94] Morch A., *Designing for radical tailorability: coupling artifact and rationale*, Knowledge-Based Systems, Vol. 7., n° 4, Butterworth-Heinermann Ltd, 1994

[MOR 97] Morch A., *Method and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach*, part 1, Dr. Scient. Thesis Research Report 241, University of OSLO, Department of Informatics, 1997

[MQS 02] *MQSeries Workflow*, http://www-3.ibm.com/software/ts/mqseries/workflow/

[MUL 97] Muller Pierre-Alain (1997). *Modélisation objet avec UML*. Edition Eyrolles

[NAR 96] Nardi B. A., *Context and consciousness : activity theory and Human-ComputerInteraction.* Eds., Cambridge, Ma : MIT Press, 1996

[OBI 02] OMG Background Information, http://www.omg.org/news/about/index.htm

[OBJ 02] http://www.markv.com/markv.com/objectmaker.htm

[OBU 02] Objecteering/UML, http://www.objecteering.com/us/index.php

[OLE 00] O'Leary D. E., *Virtual organizations*, Proceedings of the international conference on Information systems December 2000

[OMA 00] O'Malley, Jr., John Richard, *Electronic Data Interchange: An Inventory Perspective of Its Economic Viability and Recommendations for Information Technology Driven Implementation* , PhD dissertation, 2000.

[OMA 02] *OMG's Object Management Architecture (OMA)*, http://www.omg.org/gettingstarted/specintro.htm#OMA

[OMG 00] Object Management Group, *Meta-Object Facility*, 2000, OMG formal/2000-04-03.

[OMG 02] *OMG Background Information*, http://www.omg.org/news/about/

[OMG 02b] Object Management Group, *CORBA Components*, 2002, Version 3.0, OMG formal/02-06-65.

[OMG 99] Object Managment Group (1999). Meta Object Facility (MOF) Specification (Version 1.3 RTF), Appendix C : MODL Description of the MOF

[OZS 99] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999

[PAE 98] Paepcke A., Chang C-C K., Garcia-Molina H & Winograd T., Interoperability for Digital Libraries Worldwide, Communications of the ACM, April 1998/vol 41 n°4, p33-43

[PAR 01] Cécile Paris, Jean-Claude Tarby, Keith Vander Linden, *A Flexible Environment for Building Task Models,* Proceedings of Human Computer Interaction conference HCI 2001

[PET 98] Peter Y., *Gestion de la mobilité dans les environnements de communication à objets*, Th. De Doctorat en Informatique, UFR des Sciences et Techniques de l'Université de Franche-Comté, 1998, n° 704.

[PIC 97] *PICSRules 1.1*, W3C Recommendation 29 Dec 1997, http://www.w3.org/TR/REC-PICSRules

[RAV 99] Pierre-Guillaume Raverdy and Rodger Lea, *Reflection Support for Adaptive Distributed Applications*, EDOC 99, Mannheim

[RAY 01] Kerry Raymond, CiTR/DSTC, *The Fundamental Interconnectedness of All Things*, document interne

[RDF 99] *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation 22 February 1999, http://www.w3.org/TR/REC-rdf-syntax/

[REE 96] Reenskaug T., Wold P., Lehne O., *Working with Objects. The OOram Software Engineering Method*, Mannig Publications Co, 1996, 366 p

[REV 01] N. Revault, X. Blanc & J-F. Perrot, *Traduction de méta-modèles*, Langages et Modèles à Objets (Lmo'01), Vol 7 - n° 1-2/2001, R. Godin & I. Borne (ed), L'Objet - logiciel, bases de données, réseaux, pp. 95-111, Le Croisic, France, Janv (29-31), 2001, Hermès Science Publications, Paris

[REV 95] N. Revault, H.A. Sahraoui, G. Blain & J.-F. Perrot, *A Metamodeling technique: The MétaGen system*, Tools 16: Tools Europe'95, pp. 127-139, Versailles, France, Mar., 1995, Prentice Hall

[REV 96] Revault N., *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MÉTAGEN et les frameworks)*, Th. De Doctorat en Informatique, Université de Paris 6, 1996.

[RHO 00] Chris Vander Rhodes, Malissa Williams, *Competing Data Warehousing Standards to Merge in the OMG*, OMG Press, 25 Septembre 2000, http://www.omg.org/news/releases/pr2000/2000-09-25a.htm

[RIE 01] Riehle D., Fraleigh S., Bucka-Lassen D., Omorogbe N., 2001, *The architecture of a UML virtual machine.* in ACM SIGPLAN Notices, v.36 n.11, p.327-341, 11/01/2001

[SAI 01] Saikali K. *Mise en oeuvre de la Flexibilité des Workflows par l'approche Objet : 2Flow, un framework pour Workflows flexibles*, Th. De Doctorat en Informatique, École Centrale de Lyon, 2000.

[SAI 01] Saikali K. *Mise en oeuvre de la Flexibilité des Workflows par l'approche Objet : 2Flow, un framework pour Workflows flexibles*, Th. De Doctorat en Informatique, École Centrale de Lyon, 2000.

[SER 96] Sérieyx H, Azoulay H., *Face à la complexité: Mettez du réseau dans vos pyramides*. Éditions Village Mondial, Paris 1996.

[SHE 90] A. P. Sheth & J. A. Larson, *Federated database systems for managing distributed, heterogeneous and autonomous databases*, ACM Computing Surveys 22 (September 1990), 183--236

[SHE 97] A. Sheth. *From Contemporary Workflow Process Automation to Dynamic Work Activity Coordination and Collaboration*, Siggroup Bulletin, 18(3):17-20, 1997

[SIE 98] Jon Siegel, OMG overview: CORBA and the OMA in enterprise computing, Communications of the ACM, v.41 n.10, p.37-43, Oct. 1998

[SIT 89] SITPRO. *The EDIFACT service from SITPRO.* EDI standards section publication, *Simplification of Trade Procedures Board*, London, 1989. (via http://www-ensimag.imag.fr/cours/Exposes.Reseaux/EDI/edi-gene.html)

[SOA 02] *Project: SOAP to CORBA bridging software*, http://sourceforge.net/projects/soap2corba/ voir aussi http://soap2corba.sourceforge.net/

[SOL 02] Solution1Alliance, *What do we do? Control Medical Documents*, http://www.elliottdata.com/pdf/S1A_What%20Soln1Alliance%20Does.pdf

[STE 01] J. Steel, K. Raymond, *Generating Human-Usable Textual Notations for Information Models*. EDOC 2001, IEEE, Seattle, Washington USA 4-7 Septembre, Proceedings

[STR 96] Norbert A. Streitz , Heinz-Dieter Böcker, *Research on human-computer interaction and cooperative hypermedia at GMD-IPSI*, Proceedings of the CHI '96 conference companion on Human factors in computing systems : common ground: common ground, p.143-144, April 13-18, 1996, Vancouver, British Columbia, Canada

[SUB 00] Mani Subramani Eric Walden, *Economic returns to firms from business-to-business electronic commerce initiatives: an empirical examination*, actes de la 21ème conference internationale conference sur les systèmes d'information (ICIS), 2000, Brisbane, Queensland, Australia, pp 229-241

[SWE 95] Swenson, K.D. and Irwin, K., *WorkflowTechnology: Tradeoffs for Business Process Re-engineering*, in Proceedings of COOCS'95, ACM (Aug. 1995), pp. 22--29

[TAN 01] E. Tanter et N.M.N. Bouraqadi-Saâdani, *Reflex - Towards an Open Reflective Extension of Java*, In A. Yonezawa, and S. Matsuoka editors, Reflection'2001, volume 2192 of Lecture Notes in Computer Science, pages 25-43, Septembre 2001. Springer-Verlag.

[TAR 97] Tarumi, H., Kida, K., Ishiguro, Y., Yoshifu, K., and Asakura, K.: *WorkWeb System --- Multi-Workflow Management with a Multi-Agent System*, Proceedings of ACM International Conference on Supporting Group Work (Group'97), pp. 299-308 (Nov. 1997)

[TIC 97] W. F. Tichy. *A catalogue of general-purpose software design patterns*. In TOOLS USA 1997.

[TOK 02] *TokTok Download Page*, http://www.dstc.edu.au/Research/Projects/Pegamento/TokTok/download.html

[TRE 94] Jonathan Trevor , Tom Rodden , John Mariani, *The use of adapters to support cooperative sharing*, Proceedings of the conference on Computer supported cooperative work October 1994

[UDD 02] *UDDI Executive White Paper, uddi.org*, http://www.uddi.org/pubs/UDDI_Executive_White_Paper.PDF Voir http://www.uddi.org/ et http://www.uddi.org/faqs.html

[UML 01] Object Management Group, *Unified Modeling Language (UML)*, 2001, OMG formal/2001-09-67.

[UML 02] *Introduction to OMG's Unified Modeling Language*, http://www.omg.org/gettingstarted/what_is_uml.htm

[UMT 02] UML Task Force http://www.jcp.org/aboutJava/communityprocess/review/jsr040/

[VAN 95] W.M.P. van der Aalst. *A class of Petri net for modeling and analyzing business processes*. Computing Science Reports 95/26, Eindhoven University of Technology, Eindhoven, 1995

[VAN 02] Vantroys T., Peter Y.. *Un système de workflows flexible pour la formation ouverte à distance*. TICE 2002, 13 - 15 novembre 2002, Lyon, France.

[VAN 02b] Vantroys T., Rouillard J., *Workflow and Mobile Devices in Open Distance Learning*. IEEE International Conference on Advanced Learning Technology (ICALT 2002), 9 - 12 septembre 2002, Kazan, Tatarstan, Russia, pages 123 - 127.

[VIE 98] Viéville C., Derycke A., *Self-Organised Group Activities Supported by Asynchronous Structured Conversations*, Proceedings of the IFIP conference on "Virtual Campus: trends for higher education, and training", Madrid, Spain, November 1997, F. Verdjo, G.Davies (Eds), Chapmann & Hall, London, 1998, pp 191-204.

[VIN 98] Steve Vinoski, *New features for CORBA 3.0* ,Communications of the ACM October 1998, Volume 41 Issue 10

[WAR 02] Warmer J., Objecten K., 2002, *The future of UML.*

[WEG 00] A. Wegmann, G. Genilloud, *The Role of "Roles" in Use Case Diagrams*, Proceedings of 3rd International Conference on the Unified Modeling Language (UML2000), York, UK, October 2000, pp.

[WEI 01] Aaron Weiss, *Microsoft's .NET: platform in the clouds*, netWorker - ACM Press, Volume 5 , Issue 4 (December 2001), Pages: 26 – 31

[WFM 99] Workflow Management Coalition, *Terminologie et Glossaire Workflow*, Février 1999, http://www.wfmc.org

[WIE 92] Wiederhold G, Mediators in the architecture of future information systems, IEEE Computer,25:38-49, 1992.

[WIL 86] Michael Willett, *The IBM token-ring—a functional perspective*, Proceedings of 1986 fall joint computer conference on Fall.

[WMF 00] OMG, "*Workflow Management Facility Specification, v1.2*", OMG Document formal/2000-05-02, April 2000, http://www.omg.org/cgi-bin/doc?formal/2000-05-02.

[XMI 02] XML Metadata Interchage (XMI) v 1.2 http://www.omg.org/technology/documents/formal/xmi.htm

[XTK 02] *alphaWorks - XMI Toolkit*, http://www.alphaworks.ibm.com/tech/xmitoolkit

[ZOP 02] *Zope Workflow Proposal*, http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/Zope3Workflow

[ZUR] zur Muehlen, Michael; Allen, *Workflow Management Coalition Classification: Embedded and Autonomous Workflow.* WfMC White Paper. Lighthouse Point, March 10th, 2000

Table des figures

Index

W

X