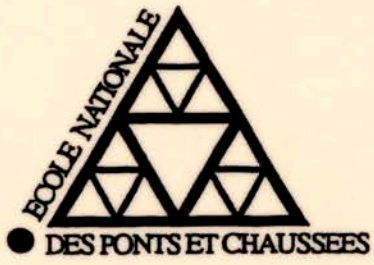
Intégration de sources de données hétérogènes

Olivier Jautzy



Thèse de Doctorat

Spécialité

INFORMATIQUE

présentée à

L'ECOLE NATIONALE DES PONTS ET CHAUSSEES

par

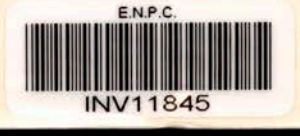
Olivier JAUTZY

Titre :

Intégration de sources de données hétérogènes : Une Approche Langage

Soutenue le 23 Mars 2000 devant le jury composé de :

M.	**Claude Delobel**	Président du jury
Mme	**Danielle Boulanger**	Directeur
M.	**Franck Lebastard**	Co-directeur
M.	**Claude Delobel**	Rapporteurs
M.	**Patrick Valduriez**	
Mme	**Mireille Blay-Fornarino**	Examinateurs
M.	**Franck Lebastard**	



X

Thèse de Doctorat

Spécialité

INFORMATIQUE

présentée à

L'ECOLE NATIONALE DES PONTS ET CHAUSSEES

par

Olivier JAUTZY

Titre :

Intégration de sources de données hétérogènes : Une Approche Langage

Soutenue le 23 Mars 2000 devant le jury composé de :

M.	**Claude Delobel**	Président du jury
Mme	**Danielle Boulanger**	Directeur
M.	**Franck Lebastard**	Co-directeur
M.	**Claude Delobel**	Rapporteurs
M.	**Patrick Valduriez**	
Mme	**Mireille Blay-Fornarino**	Examinateurs
M.	**Franck Lebastard**	

À mes parents,

À mon épouse Laurine.

Remerciements

Une thèse n'est pas seulement l'aboutissement du travail d'une seule personne, mais aussi le résultat d'une expérience et d'une vision des choses formées au fur et à mesure de rencontres, de conseils, de discussions, d'idées émises ici ou là.

Je voudrais donc remercier l'ensemble des personnes qui m'ont accompagné sur ce chemin difficile qui conduit à la rédaction d'un mémoire de thèse, en commencant par René, qui m'a fait véritablement découvrir et apprécier le monde de la recherche (en passant par l'astronomie et la diffraction des RX), et par Franck qui m'a bien sûr soutenu et conseillé depuis le début mais qui surtout m'a donné la possibilité de réaliser mon premier rêve : *me plonger dans ce monde de recherche*, et le suivant ...

Je tiens aussi à remercier tout particulièrement mon directeur de thèse, Danielle Boulanger, ainsi que Claude Delobel sans qui ce mémoire n'aurait certainement pas la qualité scientifique et la rigueur mathématique qu'il présente actuellement. Mes remerciements vont aussi à Patrick Valduriez pour m'avoir fait l'honneur d'être rapporteur scientifique de cette thèse et à Mireille Blay-Fornarino pour avoir accepté de faire partie de mon jury ainsi que pour l'ensemble des remarques qui ont contribué à améliorer ce document.

Bien entendu, mille mercis à tous ceux qui ont partagé mon séjour au CERMICS du bas : Vincent, Eric, Blaise, Fabrice, Thierry, Alain, Nicolas, Bertrand (t'inquiète pas Stéphane, je te garde une place spéciale !) et du haut : Guillaume, Gilles, Malika, Serge, et qui ont grandement contribué à l'ambiance formidable qui y règne.

Je souhaiterais aussi exprimer toute ma gratitude à Thiep Doanh qui m'a guidé sur les voies de l'informatique dans un ministère dédié aux Travaux Publics.

Enfin, il ne me vient pas de mots assez forts pour te remercier, Stéphane. Tes conseils et nos longues discussions m'ont permis très souvent de faire les bons choix, et ce dans de nombreux domaines, et de ne pas me décourager ...

"— Ceux qui prétendent détenir la vérité sont ceux qui ont abandonné la poursuite du chemin vers elle. La vérité ne se possède pas, elle se cherche."

A. Jacquard.

Table des matières

Introduction

Les *Systèmes de Gestion de Bases de Données*[1] (SGBD) ont connu de nombreuses évolutions et révolutions, liées en particulier aux formalismes de représentation et de structuration des données.

La plus récente de ces évolutions concerne le passage du *formalisme relationnel* [Codd70], structurant les données sous la forme de tables, au *formalisme objet.* Cette évolution est une conséquence de l'avènement, dans les années 90 des langages de programmation à objets tels que C++ (voir [Stroustrup93]) ou Java (voir [Gosling et al.96]). Elle est constituée de deux approches :

- Il s'agit, pour les adeptes de la première approche, de faire évoluer l'existant, c'est-à-dire de continuer à bénéficier de l'extraordinaire développement tant théorique que pragmatique du formalisme relationnel tout en adoptant certains des principes fondateurs des langages à objets que sont l'héritage, les méthodes, et l'imbrication des entités. C'est l'option suivie par les principaux éditeurs de SGBD relationnels à travers un effort de normalisation ISO/ANSI (voir [ISO95]) dans le but de proposer un standard nommé SQL3.
- La seconde approche concerne la création de SGBD structurés suivant un formalisme objet. Dans ce cas, une grande partie des recherches et des développements effectués sur le formalisme relationnel n'est pas réutilisable, mais en contrepartie, les systèmes résultants ont une capacité de modélisation accrue et peuvent aisément être utilisés par le biais des langages à objets. Là aussi, un effort de normalisation est en cours dans le but de proposer un standard nommé ODMG (voir [Cattel et al.97]).

Une des particularités du monde des bases de données est la coexistence des nombreux formalismes issus de chaque évolution. En particulier, les formalismes réseau [Tailor76], hiérarchique [Tsichritzis et al.76] et bien évidemment relationnel et objet sont tous utilisés aujourd'hui pour structurer et représenter les données. Il est à parier que même si des efforts normatifs tels que SQL3 ou ODMG sont en cours, cette caractéristique risque de perdurer. Ceci est d'autant plus vrai que les informations dites semi-structurées, disponibles notamment sur l'Internet, comme par exemple les pages au format HTML (voir [Graham99]), XML (voir [W3C99]), ou SGML (voir [ISO86]), ou, les fichiers LaTeX (voir [Lamport94]), ou bien des données techniques au format STEP (voir [Arbouy et al.94]), ou encore des données multimédia, forment une masse d'information gigantesque mais ne sont que peu prises en compte par ces nouveaux standards.

De plus, les langages de requêtes permettant d'interroger et/ou de modifier les informations contenues dans ces différentes sources de données et les outils garantissant la sécurité et la

[1] Pour une description complète de ces systèmes et des techniques qui leur sont associées, le lecteur pourra se référer à [Gardarin89], [Gardarin93] et [Gardarin99].

cohérence des données ont, à l'instar des formalismes de structuration des données, suivi de nombreuses évolutions et sont aujourd'hui tout aussi hétérogènes.

Malgré la « mondialisation de l'information », cette hétérogénéité des données, des formalismes selon lesquels elles sont structurées, des systèmes dans lesquels elles sont stockées et des outils à l'aide desquels elles sont gérées, nommée *hétérogénéité de stockage des données*, rend difficile voir impossible l'exploitation et le traitement simultané de toute l'information disponible. Pour résoudre ce problème, il est nécessaire de proposer de nouveaux outils capables de fournir un accès homogène à l'ensemble des données qui leur sont accessibles. Pour circonscrire la complexité de réalisation de tels outils, la recherche s'oriente vers la définition de Systèmes de Gestion Multibases de Données (SGMB) qui permettent d'intégrer diverses sources de données hétérogènes et donnent l'illusion d'un seul et même SGBD (voir [Silbershatz et al.95] et [Kim95b]). De tels systèmes constituent des sortes de SGBD virtuels dont les données ne sont plus stockées dans une ou plusieurs bases de données qui leur sont propres, mais au contraire distribuées et réparties dans diverses bases de données distantes, hétérogènes, autonomes et indépendantes.

Bien qu'étant un progrès majeur, les SGMB ne répondent que partiellement aux besoins de leurs utilisateurs. En effet, les SGMB offrent par essence un accès homogène mais identique pour l'ensemble de ses utilisateurs. Ceux-ci sont donc contraints d'utiliser les outils et les techniques offerts par le SGMB, qui ne correspondent pas forcément à leurs besoins, alors même que ceux disponibles au sein des sytèmes stockant réellement les données seraient adaptés à leurs besoins. Illustrons cela à travers le concept de transaction. Les transactions ont initialement été introduites dans les SGBD puis ont été généralisées à tous les systèmes d'informations pour garantir la cohérence des informations qu'ils contiennent en cas de défaillance ou d'accès concurrents aux données. Les transactions peuvent être utilisées pour garantir la cohérence des données dans le cadre d'applications « classiques » de gestion d'informations, comme par exemple les systèmes bancaires ou les systèmes de réservation des compagnies aériennes. Il a aussi été démontré l'utilité de disposer d'une couche transactionnelle dans des systèmes allant des langages de programmation distribués tels qu'Argus [Liskov88] ou Avalon/C++ [Detlef et al.88] jusqu'aux systèmes d'exploitation tels que Tabs [Spector85] ou Camelot [Eppinger et al.86] en passant par les outils de CAO/DAO. Les systèmes transactionnels ont, à l'origine, été conçus et optimisés pour les applications « bases de données classiques » et dans le but de ne manipuler, sur une courte durée, que des informations simples stockées dans ces bases. Or, les applications complexes telles que les outils de CAO/DAO mettent en œuvre sur de longues durées des données beaucoup plus complexes et doivent être basées sur des systèmes transactionnels plus évolués. Il n'existe donc pas une seule sémantique transactionnelle mais plusieurs sémantiques transactionnelles distinctes adaptées à chaque domaine d'application. Le choix de l'outil transactionnel nécessaire au développement d'une application doit ainsi être guidé par le type de celle-ci plutôt que par les données et les systèmes de stockage disponibles. Les utilisateurs des SGMB souhaitent donc, comme tout utilisateur de système d'information, mettre en œuvre les techniques et les modèles transactionnels adaptés à leurs besoins, ceux-ci ne pouvant être déterminés qu'en fonction du type d'application qu'ils souhaitent développer et/ou utiliser. Or le modèle et les techniques transactionnels mis en œuvre au sein d'un SGMB sont nécessairement fixés indépendamment des besoins de leurs utilisateurs, qui peuvent être hétérogènes et même évoluer dans le temps.

En résumé, les SGMB ne prennent pas en compte l'hétérogénéité des besoins de leurs utilisateurs en termes d'outils et de techniques de gestion des données, nommée *hétérogénéité d'accès*

aux données. Ceci induit que lorsque l'on souhaite développer des applications exploitant des données stockées au sein de systèmes hétérogènes et indépendants il faut actuellement choisir entre bénéficier d'un accès homogène à ces données et gérer ces données selon ses besoins.

A cet égard, il nous semble intéressant de proposer un nouvel outil permettant de résoudre simultanément les problèmes liés à l'hétérogénéité de stockage des données et à l'hétérogénéité d'accès aux données. Nos recherches ont consisté à modéliser un langage de programmation basé sur un formalisme objet réflexif et sur une architecture de type SGMB. Notre système permet de manipuler et de gérer à l'identique et suivant les besoins de l'utilisateur des données volatiles ou persistantes, ces dernières pouvant être stockées dans des sources de données distribuées et hétérogènes. Notre approche a été validée par l'implantation d'une extension au langage Java.

Nos travaux sont décrits dans ce mémoire en suivant le plan présenté ci-après.

Dans le chapitre 1, nous présenterons l'état de l'art en matière d'intégration de sources de données hétérogènes. Nous décrirons les différents systèmes et modélisations proposés dans la littérature. Durant cette étude, nous porterons plus particulièrement notre attention sur l'architecture des SGMB et sur les problèmes transactionnels, de gestion de requêtes ou d'intégration structurelle et sémantique qu'ils soulèvent.

Le chapitre 2 expose les objectifs de nos travaux et les caractéristiques générales des résultats de nos recherches. En particulier, nous détaillerons les concepts d'objet transactionnel, d'envoi de message transactionnel, de vue et de comportement.

Dans le chapitre 3, nous décrivons tout d'abord le système transactionnel résultant de nos recherches en nous basant, d'une part, sur le modèle formel introduit dans le chapitre 2 pour la partie architecturale du système et, d'autre part, sur une extension du formalisme graphique défini par Gray (voir [Gray et al.93]) pour la partie comportementale du système. Nous présenterons aussi des modèles et des techniques dédiés à la gestion de la persistance, à l'évaluation de requêtes au sein de notre système et à la restructuration virtuelle de données.

Pour terminer ce mémoire, nous consacrerons un dernier chapitre à la description de l'implantation de notre système en Java. Nous verrons notamment comment ajouter un niveau réflexif suffisant à ce langage pour garantir une implantation optimale de nos modèles, ainsi que les restrictions actuelles d'utilisation de notre système.

Note de Vocabulaire. Nous avons utilisé jusqu'à présent le terme de *formalisme* pour désigner l'ensemble des concepts permettant à l'utilisateur d'un langage ou d'un SGBD de décrire la structure, la manipulation et le contrôle de ses données, c'est-à-dire, une abstraction du monde réel souvent nommée *schéma* ou *schéma conceptuel* dans le domaine des bases de données ou *modèle* dans le domaine des langages.

Dans ce dernier domaine, l'ensemble des capacités d'expression d'un langage, soit son formalisme est souvent défini à l'aide du langage, c'est-à-dire sous la forme d'un modèle, suivant le terme consacré dans ce domaine. Ainsi, ce que nous nommions formalisme peut être décrit comme un modèle de modèle, appelé *métamodèle* .

Comme l'objet de nos travaux est la modélisation d'une extension d'un langage, nous emploierons dans la suite de ce mémoire le terme de métamodèle en lieu et place de formalisme.

Première partie

Des convertisseurs aux langages réflexifs à objets

Chapitre 1

Des convertisseurs aux SGMB

Pour résoudre les problèmes liés à l'intégration de sources de données hétérogènes, la recherche s'est orientée vers la définition de *Systèmes de Gestion Multibases de Données* (SGMB), sortes de SGBD virtuels au dessus de sources de données hétérogènes, donnant l'illusion d'un seul et même SGBD (voir par exemple [Ahmed et al.91], [Kelley et al.95], [Kleewein96], [Kuno et al.93], [Litwin et al.90], [Thomas et al.90], ...).

Ce chapitre est consacré à l'étude de ces systèmes. Après un bref historique des solutions proposées dans la littérature pour résoudre les problèmes liés à l'hétérogénéité de stockage des données, nous commencerons notre étude par la présentation d'une taxonomie et des objectifs principaux des SGMB. Nous décrirons ensuite l'architecture générale de ces systèmes avant de passer en revue leurs différentes composantes et les techniques à mettre en œuvre pour garantir l'illusion d'un seul et même SGBD.

Cette dernière partie permettra au lecteur de se familiariser, en premier lieu, avec les problèmes de gestion transactionnelle liés à l'autonomie des sources de données locales. En particulier, nous aborderons les notions de verrous mortels, d'atomicité et de sérialisabilité et nous présenterons les différentes solutions proposées dans la littérature pour résoudre ou contourner les problèmes associés à ces concepts. Nous introduirons ensuite différentes techniques utilisées pour construire et présenter à l'utilisateur des données virtuelles. Enfin, nous décrirons comment décomposer des requêtes émises par l'utilisateur d'un SGMB en requêtes compréhensibles par les sources de données locales associées à celui-ci.

Dans ce chapitre, nous présenterons aussi brièvement les problèmes liés aux besoins d'intégration sémantique automatique ou semi-automatique de sources de données hétérogènes.

1.1 Du plus simple au plus complexe

Les premiers véritables besoins d'interopérabilité dans le monde des bases de données sont apparus en même temps que le métamodèle relationnel. Il s'agissait alors de faciliter la migration d'un parc de SGBD obsolètes, dont les données étaient structurées suivant les métamodèles hiérarchique ou réseau, vers un parc de SGBD relationnels. Des outils dits *convertisseurs* ont été développés dans ce but. Plus tard, ceux-ci ont laissé la place aux *passerelles* qui permettent

de conserver les données au sein des SGBD obsolètes tout en les exploitant à l'aide d'un nouveau métamodèle. Récemment, à ces besoins de migration des données d'un SGBD dans un autre et de transformation d'un métamodèle à un autre se sont ajoutés des besoins d'intégration de plusieurs sources de données hétérogènes. Deux familles d'outils en résultent : les *entrepôts de données* et les *systèmes de gestion multibase de données*. La première correspond à une évolution des convertisseurs alors que la seconde correspond plutôt à une évolution des passerelles. Dans la suite, nous présentons plus précisément chacun de ces outils.

1.1.1 Les convertisseurs

Le rôle de ces systèmes est de convertir toutes les données d'un SGBD A dans un autre SGBD B. Il s'agit ici de réaliser une copie des données de A dans B tout en transformant leurs structures du formalisme du système A au formalisme du système B. Cette approche est notamment celle choisie par IBM pour leur logiciel IMS Extract qui permet de convertir des données stockées au sein d'une base IMS en données relationnelles.

En règle générale, cette solution est utilisée lorsque le système A est amené à disparaître. Dans le cas contraire, de nombreux problèmes se posent. En particulier, une conséquence de la coexistence des données originales et de leur copie est le risque de fortes incohérences entre le système de départ et le système d'arrivée. En effet, si les mêmes données sont modifiées de façon différente dans les deux systèmes, ces données ne sont plus cohérentes entre elles. Plus généralement, quelle que soit la politique de mise à jour des données du système A vers le système B pour prendre en compte les modifications apportées directement sur A, et, quelle que soit la politique de mise à jour des données de B vers A pour prendre en compte les modifications apportées directement sur B, comme les données existent physiquement dans les deux systèmes, ces derniers ne sont cohérents qu'à la suite immédiate d'une mise à jour. Enfin, pour en revenir au sujet général de ce mémoire, cette solution augmente le nombre de données disponibles car ces dernières sont copiées lors de la conversion. Par conséquent, les convertisseurs contribuent en un sens à l'augmentation de l'hétérogénéité des données !

1.1.2 Les passerelles

Une solution a été proposée pour pallier les défauts des convertisseurs. Au lieu de copier les données d'un système à un autre, il s'agit de transformer une requête émise dans le langage du système B en une ou plusieurs requêtes compréhensibles par le système A, puis de transformer les données résultats, structurées suivant le métamodèle du système A, en données structurées suivant le métamodèle du système B. Ainsi, les données du système A sont vues suivant le métamodèle de B comme précédemment, cependant, dans ce cas, comme aucune copie de données n'est effectuée[1], les problèmes décrits en 1.1.1 sont résolus.

Cette approche a été choisie notamment par la société Oracle pour son système *CONNECT . Ce système est dédié à la transformation de données des métamodèles réseau ou hiérarchique au métamodèle relationnel.

[1] Notons que, dans le cas des passerelles, une copie est tout de même effectuée puisqu'il faut présenter les données à l'utilisateur du système B. Ce n'est cependant qu'une copie temporaire qui n'entraîne, en règle générale, aucune incohérence entre les deux systèmes.

1.1.3 Les entrepôts de données

Les deux solutions présentées précédemment ne sont pas adaptées à l'intégration de sources de données hétérogènes. En particulier, les passerelles ou les convertisseurs, dédiés à l'exploitation d'une seule base de données, ne possèdent pas de capacité d'homogénéisation des structures ni celle d'assurer des propriétés transactionnelles minimales lors d'accès à des données provenant de plusieurs sources distinctes. Par exemple, supposons que nous souhaitions intégrer deux bases de données A et B comportant chacune des informations sur des comptes bancaires mais structurant ces informations de manière distincte. Il n'est pas possible, à l'aide d'un convertisseur ou d'une passerelle, d'envoyer une unique requête au système permettant de récupérer des résultats homogènes provenant simultanément de A et de B. De même, si l'on souhaite effectuer une opération de transfert (débit/crédit) d'argent depuis un compte stocké dans A jusqu'à un compte stocké dans B, le système ne peut en assurer l'atomicité, c'est-à-dire qu'il est possible que seul le débit (ou le crédit) impliqué dans l'opération soit effectué.

Une solution permettant de résoudre ce problème est contenue dans l'approche *entrepôts de données* (voir [Widom95]). Un entrepôt de données est une sorte de SGBD, couplé avec plusieurs convertisseurs, qui permet d'intégrer des données stockées au sein de plusieurs SGBD distincts et hétérogènes. Les données construites à l'aide de l'entrepôt de données sont utilisées généralement pour faciliter l'extraction d'informations dans le but de réaliser des analyses statistiques, ce que l'on connaît plus souvent sous l'expression anglaise « Data Mining ». Elles ne nécessitent souvent qu'un accès en lecture, ce qui supprime les principaux problèmes transactionnels. Cependant, il subsiste une partie des problèmes décrits en 1.1.1 liés aux mises à jour des données de l'entrepôt de données à partir de celles des SGBD. En effet, comme les données sont copiées, il existe un intervalle de temps durant lequel les données de l'entrepôt n'évoluent plus alors que celles des SGBD peuvent changer. Pour préserver la cohérence des données de l'entrepôt, au bout de chaque intervalle, une opération de mise à jour est exécutée. Durant cette opération, l'entrepôt de donnée est, en règle générale, inaccessible[2]. La définition de l'intervalle de temps est donc un point critique de la mise en place d'un entrepôt de données. S'il est trop long, les informations contenues dans l'entrepôt peuvent ne plus avoir de sens (par exemple un intervalle de un mois pour le solde d'un compte !) Dans le cas contraire, s'il est trop court, les opérations de mises à jour, qui supposent l'indisponibilité du système, seront trop longues par rapport au temps disponible pour les utilisateurs (voir [Staudt et al.96]).

1.1.4 Les systèmes de gestion multibases de données

Pour résoudre le problème d'intégration de sources de données hétérogènes de façon optimale, la recherche s'est orientée vers les Systèmes de Gestion MultiBases (SGMB). Le reste de ce chapitre est consacré à la présentation de ces systèmes. Nous commencerons par les présenter de manière générale notamment à travers une taxonomie et des objectifs. Puis nous rentrerons dans le détail des différents concepts à mettre en œuvre pour résoudre efficacement les problèmes d'intégration de sources de données hétérogènes.

[2] Le plus souvent pour éviter les problèmes transactionnels.

1.1.4.1 Taxonomie

De nombreuses classifications des SGMB ont été données dans la littérature (voir par exemple, [Tresch et al.94]). Nous décrivons ci-dessous celle présentée dans [Sheth et al.90] qui tout à la fois définit les SGMB et présente une taxonomie des différents systèmes.

Un SGMB est un ensemble de SGBD ou, plus généralement, de sources de données, présentant à un utilisateur l'illusion d'un seul et même système de gestion de bases de données. Cette définition, très générale, regroupe un certain nombre de systèmes distincts. Pour en préciser les différences, nous reprenons ci-dessous la taxonomie décrite en [Sheth et al.90] (voir la figure 1.1)

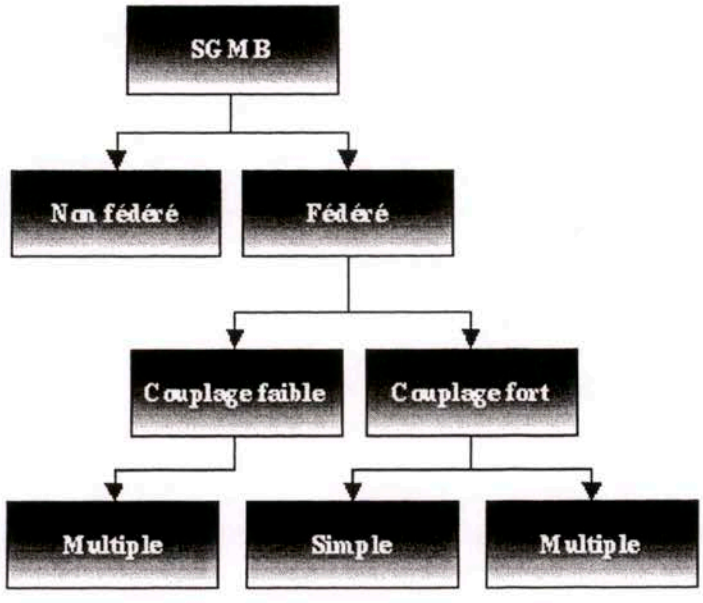


FIG. 1.1 – Taxonomie des SGMB

Dans la suite, nous considérons que chaque source de données élément du SGMB est appelée *SGBD local* ou, plus généralement, *source de données locale*. Nous appellerons *modèle local*[3], le schéma ou modèle d'une source de données locale, c'est-à-dire l'ensemble des structures permettant de stocker les données au sein du SGBD. Notons que le modèle local est exprimé suivant le métamodèle de la source de données locale. Enfin, nous appellerons *modèle global* le modèle du SGMB. Notons que ce modèle est exprimé dans le métamodèle du SGMB et qu'il est le plus souvent incomplet. Toutes les données et les structures des sources locales ne sont pas forcément d'équivalent global).

- Un SGMB est dit *fédéré* si les sources de données locales sont autonomes, c'est-à-dire si elles respectent les critères d'autonomie que nous définirons en 1.1.4.3. Dans le cas contraire, le SGMB est non fédéré. Les SGMB non fédérés sont principalement des SGBD distribués.
- Un SGMB fédéré est dit *à couplage fort* si l'administrateur ou le système est responsable de la création, maintenance du ou des modèles globaux du SGMB. Dans le cas contraire, le SGMB fédéré est dit à couplage faible.

[3]Il est possible qu'un tel modèle n'existe pas, par exemple si la source de données locale est un site Web. Cependant, pour la clarté de l'exposé, nous considérons dans ce mémoire qu'il est possible d'extraire un modèle de toutes les sources de données.

- Enfin, un SGMB fédéré est *multiple* s'il existe plusieurs modèles globaux. S'il n'y en a qu'un, le SGMB fédéré est simple.

Remarquons qu'un SGMB fédéré à couplage faible est nécessairement multiple vu que chaque utilisateur définit son propre modèle (il n'y a pas d'administrateur unique chargé de la conception d'un ou plusieurs modèles globaux). Un SGMB fédéré à couplage fort peut être soit simple soit multiple. Dans ce dernier cas, la multiplicité permet d'avoir plusieurs « petits modèles » au lieu d'un seul contenant les structures de toutes les sources de données du SGMB. Ceci permet d'une part de faciliter la maintenance du système, et d'autre part, de mettre plusieurs modèles, donc plusieurs sémantiques à la disposition des utilisateurs du SGMB.

Dans la suite de ce mémoire nous ne nous intéresserons qu'aux SGMB fédérés. Par souci de simplification, nous les nommerons simplement SGMB, ce qui est par ailleurs souvent le cas dans la littérature.

1.1.4.2 Objectifs

Pour permettre au lecteur de mieux cerner la nature des SGMB, nous définissons ci-dessous leurs principaux objectifs. Ceux-ci sont le résultat de la combinaison des besoins industriels et de la volonté de définir des SGMB pouvant prendre en compte le plus grand nombre de sources de données disponibles tout en donnant à l'utilisateur la plus grande souplesse d'utilisation. Ces objectifs ont été partiellement décrits dans [Kim95a] et [Kleewein96].

Objectif 1 Les SGMB doivent remplacer la conversion de données d'une source de données dans une autre. En d'autres termes, les SGMB doivent supprimer les besoins de convertisseurs.

Objectif 2 La mise en place d'un SGMB ne doit impliquer aucune modification des sources de données locales. En d'autres termes, chaque source de données locale ne voit le SGMB que comme une application classique.

Objectif 3 La mise en place du SGMB ne doit pas empêcher les anciennes applications ou de nouvelles applications d'avoir accès aux sources de données locales. En d'autres termes, un SGMB ne doit pas supprimer l'accès natif aux sources de données locales.

Objectif 4 La mise en place du SGMB ne doit introduire aucune modification de l'administration des SGBD locaux.

Objectif 5 L'hétérogénéité des langages de requêtes ou des modes d'accès aux sources de données locales doit être cachée aux utilisateurs et aux applications du SGMB. Chaque utilisateur, ou chaque application doit pouvoir interroger le SGMB comme un seul SGBD.

Objectif 6 L'hétérogénéité des métamodèles structurant les données des sources de données locales ne doit pas être visible pour les utilisateurs et les applications.

Objectif 7 L'hétérogénéité physique des sources de données locales, c'est-à-dire, le système d'exploitation, le protocole réseau utilisé, le type de machine, la localisation sur le réseau de la machine où se situe la source de données locale, doit aussi être cachée aux utilisateurs et aux applications.

Objectif 8 Les SGMB doivent supporter les transactions distribuées sur les divers SGBD locaux, l'hétérogénéité des modèles transactionnels des sources de données locales étant là aussi cachée aux utilisateurs et aux applications.

Objectif 9 Les SGMB doivent permettre aussi bien la lecture que la mise à jour et l'insertion des données.

Objectif 10 Les SGMB doivent être de véritables SGBD. Ils doivent disposer d'outils de sécurité, de gestion des droits utilisateurs, de reprise après panne et de réplication.

Objectif 11 Les performances des SGMB doivent être aussi proches que possible des performances d'un SGBD distribué.

1.1.4.3 Les autonomies DEC

Les objectifs définis ci-dessus peuvent être en partie décrits du point de vue des sources de données locales. Dans ce cas, il s'agit de définir l'autonomie nécessaire de celles-ci par rapport au SGMB pour respecter les objectifs précédemment cités. En fait, il est possible de définir trois types d'autonomie, que nous nommerons D,E et C ([Sheth et al.90], repris par [Breitbart et al.95]). Nous verrons plus loin que ceux-ci ont des conséquences très importantes notamment sur la gestion des transactions au sein des SGMB.

(D) Autonomie de Conception Aucune modification ne doit être apportée au logiciel des sources de données locales pour prendre en compte le SGMB. En effet, le code source de celui-ci peut ne pas être disponible, et même dans le cas contraire, des changements risquent de rompre l'intégrité du système local et donc, le SGMB ne suivrait plus les objectifs 2, 3 et 4.

(E) Autonomie d'Exécution Chaque source de données locale garde le contrôle complet le cas échéant, de l'exécution des transactions locales sur son site. Chaque source de données locale prend donc unilatéralement les décisions de sauvegarde ou non des informations dans la base, et ceci pour respecter les objectifs 2 et 4.

(C) Autonomie de Communication Les sources de données locales ne sont pas supposées communiquer entre elles, ni fournir des information d'optimisation au SGMB ce qui contredirait l'objectif 2.

1.2 Architecture des SGMB

Nous avons vu ci-dessus, les différentes propriétés que devaient vérifier les SGMB pour garantir une intégration optimale de sources de données locales. Nous décrivons dans cette partie l'architecture à mettre en œuvre dans le but de respecter au mieux ces propriétés. En fait, nous présentons ici trois architectures : celle du gestionnaire de données, celle du gestionnaire transactionnel et celle du gestionnaire de requête. Chacune représente l'une des composantes majeures des SGMB et permet non seulement de rendre compte globalement de la complexité de ces systèmes mais aussi de les décomposer en entités spécialisées, indépendantes et donc de faciliter leur étude et leur réalisation.

1.2.1 Architecture du gestionnaire de données

L'architecture du gestionnaire de données est certainement la plus représentative de la complexité des SGMB et aussi la plus spécifique à ces systèmes. Pour la définir, il est intéressant de se baser sur l'objectif 10, c'est-à-dire de partir de l'architecture du gestionnaire de données d'un SGBD classique. A cet égard, après quelques définitions, nous présenterons l'*architecture ANSI/X3/SPARC* préconisée comme architecture de référence pour les SGBD classiques. Puis nous décrirons à travers une extension de cette architecture, celle des SGMB.

1.2.1.1 Définitions

Nous listons ci-dessous quelques définitions de termes qui caractérisent les divers composants employés lors de la description des architectures que nous présentons dans les paragraphes suivants.

Les structures de données. Une *structure de données* caractérise la représentation informatique, exprimée dans un métamodèle particulier, d'un ensemble de données du monde réel. Par exemple, les tables relationnelles sont des structures de données exprimées dans le métamodèle relationnel.

Les modèles. Un *modèle* est un ensemble de structures de données exprimées dans le même métamodèle.

Les bases de données. Une *base de données* est un ensemble de données structurées suivant un modèle fixé et donc nécessairement suivant un métamodèle fixé. Une base de données doit aussi vérifier certaines propriétés, mais nous nous restreignons à cette définition suffisante pour les besoins de ce chapitre. Pour une définition plus précise, le lecteur pourra se référer à [Gardarin93].

Les requêtes. Une *requête* est une expression informatique formulée dans un langage fixé. Elle est utilisée pour filtrer un ensemble de données en fonction de certains critères. Une requête est construite en fonction du modèle structurant les données à filtrer. Le langage utilisé pour

exprimer des requêtes sur une base de données est très souvent lié au métamodèle suivant lequel est exprimé le modèle de cette base. En conséquence, si ce métamodèle est appelé A, nous appellerons A' le langage de requêtes correspondant.

Les processeurs. Un *processeur* est un module logiciel effectuant des opérations sur des données et/ou sur des requêtes. Pour la description des architectures présentées dans ce chapitre, nous utilisons les quatre types de processeurs suivants (d'après [Sheth et al.90]) :

- Les *traducteurs* permettent d'une part de traduire des requêtes exprimées dans un langage A' en requêtes exprimées dans un langage B' et, d'autre part, de convertir des données structurées suivant un modèle exprimé dans le métamodèle B en données structurées suivant le même modèle mais exprimées dans le métamodèle A.
- Les *filtres* permettent d'une part de transformer des requêtes portant sur un modèle X en requêtes portant sur un autre modèle Y, X et Y étant exprimés dans le même métamodèle, et d'autre part, de transformer des données structurées suivant Y en données structurées suivant X. En particulier, ils sont capables d'effectuer sur ces données des opérations de projection, de généralisation ou de renommage.
- Les *constructeurs* partitionnent et/ou répliquent des requêtes émises par un processeur sur plusieurs autres processeurs. Réciproquement, ils combinent les données émises par plusieurs processeurs en un ensemble de données pouvant être traité par un seul processeur. Notons que ces processeurs travaillent à métamodèle constant.
- Les *accesseurs* sont consacrés à l'exécution de requêtes au dessus d'une base de données locale.

1.2.1.2 Architecture de base à 3 niveaux

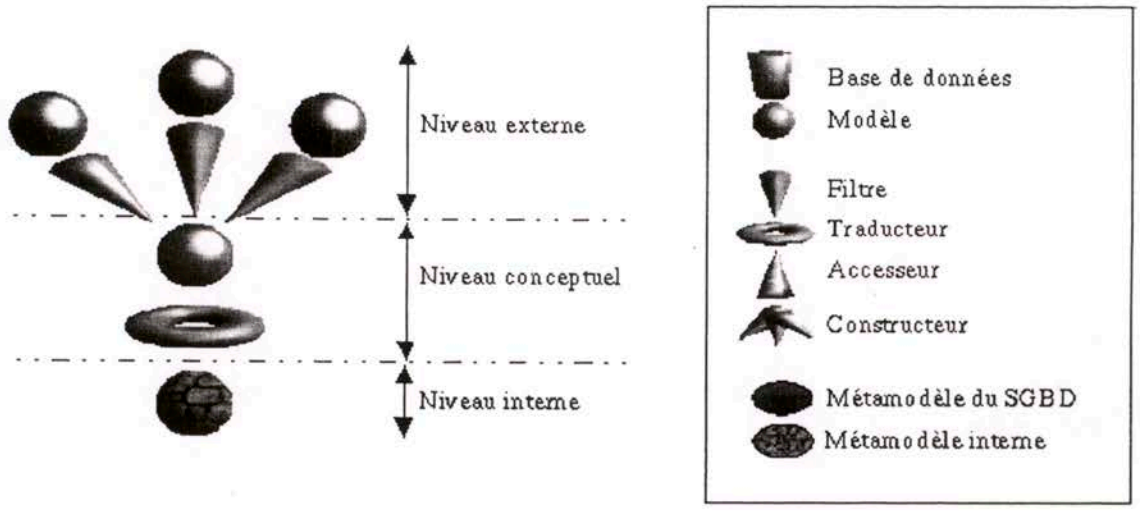


FIG. 1.2 – Architecture de base à 3 niveaux

Comme nous l'avons précisé en introduction, les outils de stockages de données ont évolué depuis la représentation des données sous la forme d'articles au sein de fichiers jusqu'à l'utilisation de métamodèles complexes. Les outils actuels présentent aux utilisateurs des données

exprimées suivant un formalisme « assez naturel ». Cependant, le stockage effectif des données est toujours sous la forme de fichiers. On voit ainsi apparaître deux niveaux caractérisant le gestionnaire de données, le *niveau physique* de stockage effectif, et le *niveau conceptuel* de présentation des données. Un gestionnaire de données structuré uniquement suivant ces deux niveaux présente quelques inconvénients. En particulier, chaque utilisateur du SGBD a accès aux mêmes données structurées suivant le même modèle. Or, pour des raisons de sécurité, il peut être intéressant de cacher une partie des données aux utilisateurs. Ou, plus simplement, certains utilisateurs ne veulent avoir accès qu'à une partie des données peut-être même structurées différemment. En effet, si l'on considère une base de données stockant des informations sur les employés d'une entreprise, par exemple le nom, le prénom, le salaire net et les primes, il est souhaitable que tous les utilisateurs de cette base n'aient pas accès aux données sur tous les employés. De plus, il est possible que certains utilisateurs souhaitent connaître le salaire des employés indépendamment des primes, alors que d'autres peuvent vouloir connaître le revenu global des employés. En conséquence, un troisième niveau est proposé dans le cadre de l'architecture définie par le groupe ANSI/X3/SPARC (voir [ANSI75] et [ANSI78]). Il s'agit d'un niveau de présentation, dit *niveau externe* dans lequel les données sont structurées suivant un modèle adapté à chaque utilisateur et exprimées dans le métamodèle associé au niveau conceptuel.

L'architecture à trois niveaux ainsi formée peut être décrite en termes de processeurs, de modèles, et de bases de données (voir la figure 1.2).

- Le niveau interne correspond au stockage physique des données. A ce niveau, celles-ci sont exprimées suivant un métamodèle interne et structurées suivant un modèle interne. Ce niveau représente les données du point de vue de la machine, par exemple, les fichiers qui les contiennent, les articles de ces fichiers, le chemin d'accès aux articles. Notons ici que ce niveau est dépendant de l'architecture matérielle sur laquelle repose la base de données.
- Le niveau conceptuel comprend un traducteur permettant de transformer les données du modèle interne au modèle, dit conceptuel, de la base de données. A ce niveau, les données sont exprimées suivant le métamodèle conceptuel, c'est-à-dire le métamodèle associé à la base de données (par exemple le métamodèle relationnel lorsque l'on parle de bases de données relationnelles). C'est le niveau central de tout SGBD. Il correspond à la structure sémantique de toutes les données de la base.
- Enfin le niveau externe est composé de filtres permettant de transformer les données du modèle conceptuel en un modèle dit externe adapté à chaque utilisateur. Notons que cette transformation se fait à métamodèle constant.

1.2.1.3 Architecture à 5 niveaux

L'architecture décrite précédemment ne prend pas en compte les spécificités des SGMB que sont la distribution, l'autonomie et l'hétérogénéité. De nouvelles architectures ont été présentées par exemple dans [Boulanger et al.98], [Fankhauser et al.96], et [Jeffery et al.90], ou en [Oszu et al.99] et [Pitoura et al.95], ou encore dans [Saltor et al.94], [Sheth et al.90] et [Tomasic et al.95]. Ces architectures, bien que différentes, présentent des similitudes. En se basant sur ces similitudes, il est possible de définir une architecture « générique » des SGMB. Cette architecture, qui est composée de cinq niveaux, est définie ci-après et présentée dans la figure 1.3. Notons que cette architecture peut aussi être vue, par généralisation, comme une architecture à trois niveaux telle que celle décrite dans la partie précédente.

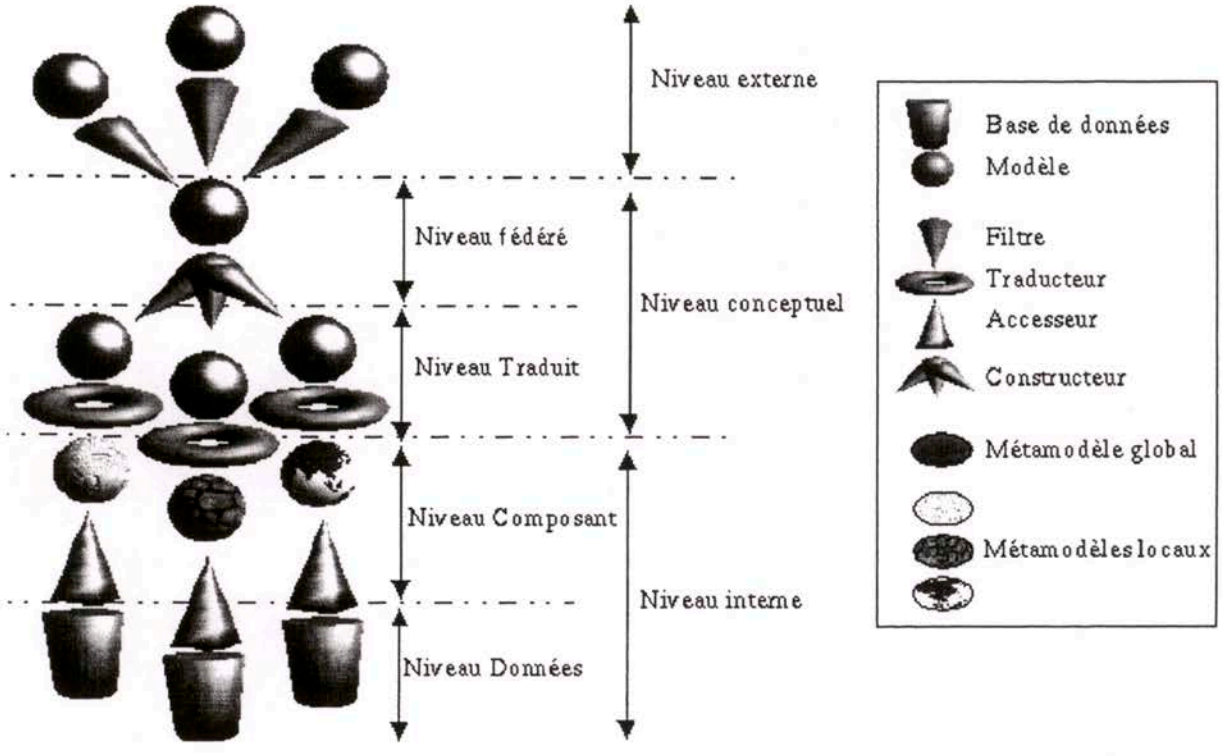


FIG. 1.3 – Architecture à 5 niveaux

Les niveaux Données et Composants. Dans le but de respecter la propriété d'autonomie de conception des SGMB, il est nécessaire de considérer les SGMB comme de simples utilisateurs de leurs sources de données locales. Si l'on souhaite définir une architecture hiérarchique à l'image de celle décrite par l'ANSI/SPARC/X3 pour les SGBD, il est possible de la baser sur un ou plusieurs modèles externes de chaque SGBD local tels que ceux présentés précédemment. Notons ici que ces modèles doivent être décrits à la fois au sein du SGBD local pour permettre la gestion physique des données et au sein du SGMB pour lui permettre d'accéder aux données stockées et de créer une base virtuelle. Alors que les sources de données locales constituent le *niveau local* de l'architecture du SGMB, la description de ces modèles, dits modèles exportés, au sein du SGMB, ainsi bien évidemment que les accesseurs permettant aux SGMB d'exploiter les données stockées, constituent le *niveau composant*. Comme les modèles exportés ne sont qu'une copie de modèles externes des SGBD locaux prenant part au SGMB, ils sont structurés suivant le métamodèle de leur SGBD d'origine.

Ces deux niveaux forment la couche basse du SGMB c'est-à-dire l'accès en mode natif des SGBD locaux. Ils doivent être donc complètement cachés à l'utilisateur à l'instar du niveau interne de l'architecture à 3 niveaux des SGBD. On peut donc considérer ces deux niveaux comme formant le niveau interne du SGMB.

La plupart des travaux portant sur les SGMB se basent sur ces deux niveaux pour décrire l'architecture de ces systèmes. Néanmoins, quelques recherches ont porté sur la définition d'un niveau supplémentaire. Il s'agit d'une part de permettre aux administrateurs des SGBD locaux de définir un modèle externe unique pour l'ensemble des SGMB souhaitant accéder aux données stockées dans ces systèmes dans le but de faciliter leur maintenance (voir [Saltor et al.94]).

Ceci peut être réalisé en complétant le niveau composant par des filtres et des modèles dits modèles exportés restreints. Les filtres permettent aux SGMB de ne voir qu'une partie du modèle exporté de chaque SGBD, celle-ci étant structurée éventuellement selon les besoins du SGMB. Par ailleurs, le rôle des modèles exportés restreints peut-être étendu par la possibilité d'y rajouter des informations sémantiques facilitant une intégration sémi-automatique au sein du SGMB (voir [Klas et al.96b]). Par souci de simplification, nous ne considérerons dans cet exposé que l'architecture présentée plus haut. Cette simplification est cependant sans perte de généralité car les accesseurs peuvent être considérés comme des processeurs ayant aussi le rôle de filtre, rendant par là même possible de confondre modèles externes et modèles externes restreints.

Les niveaux Traduction et Fédération. Pour suivre les objectifs 5 et 6, le ou les modèles du SGMB, en tant que SGBD, doivent être structurés suivant un seul métamodèle, le métamodèle global. L'étape suivante dans l'architecture d'un SGMB est donc la traduction de tous les modèles exportés de leur métamodèle d'origine dans le métamodèle global. Cette étape donne lieu, à l'aide de traducteurs, à la création des modèles dits traduits. Ces modèles et ces traducteurs forment le *niveau traduction*. Ce niveau est un niveau clé de l'architecture d'un SGMB car il assure une homogénéisation, sur le plan du métamodèle, des structures de données. Ceci suppose que la couche logicielle gérant ce niveau soit suffisamment ouverte pour permettre l'ajout de nouveaux traducteurs autorisant ainsi le SGMB à exploiter des sources de données non définies lors de la mise en place du SGMB. Cette ouverture est très importante car elle rend le SGMB peu sensible aux évolutions des métamodèles.

L'homogénéisation des structures sur le plan du métamodèle n'est pas suffisante pour répondre aux objectifs des SGMB (notamment à l'objectif 7). Il est aussi nécessaire d'homogénéiser les structures et les données sur le plan du modèle. Le SGMB doit ainsi contenir un niveau d'intégration des divers modèles traduits pour permettre de présenter aux utilisateurs un ensemble de données homogènes. Cette étape peut être réalisée en suivant diverses approches. Les principales différences entre celles-ci résident dans le nombre de modèles globaux pouvant être gérés simultanément par le SGMB. Nous avons déjà soulevé de telles distinctions dans la partie 1.4.1 lors de la présentation d'une taxonomie pour les SGMB. Il s'agit soit de donner la possibilité à chaque utilisateur de définir plus ou moins son propre modèle à la manière de [Tomasic et al.95], soit de laisser cette tâche à l'administrateur du SGMB à la manière de [Klas et al.96a], les utilisateurs se partageant alors le même modèle. Comme nous l'avons déjà précisé, on obtient respectivement des SGMB multiples ou simples. Dans tous les cas, le ou les modèles globaux sont obtenus en combinant plusieurs modèles exportés à l'aide d'un ou plusieurs constructeurs. Ces modèles et ces constructeurs forment le *niveau fédération*. Par souci de simplification, nous n'avons représenté dans la figure 1.3 qu'un seul modèle global et qu'un seul constructeur.

Enfin, notons que la combinaison des niveaux traduction et fédération peut être vue, si l'on se réfère à l'architecture en 3 niveaux, comme le niveau conceptuel du SGMB. En effet, ce niveau est structuré suivant le métamodèle global du SGMB et présente une abstraction sémantique par rapport au niveau interne, combinaison des deux niveaux données et composants, ceux-ci pouvant être considérés comme la couche interne du SGMB.

Le niveau Externe. Pour respecter l'objectif 9, il faut fournir au SGMB un niveau externe au moins similaire à celui des SGMBD tel que défini par l'ANSI/X3/SPARC [ANSI75]. Le *niveau externe* d'un SGMB est ainsi composé des filtres et des modèles externes du SGMB comme cela est représenté dans la figure 1.3.

Quelques travaux de recherche ont porté sur l'extension de ce niveau dans le cas des SGMB (voir par exemple [Saltor et al.94]). Le maître mot de ces systèmes est d'aider les utilisateurs ou applications à exploiter au mieux les données disponibles. Cela peut impliquer de présenter non seulement les données suivant le modèle de l'utilisateur ou de l'application mais aussi suivant le métamodèle souhaité par l'utilisateur ou celui à l'aide duquel est écrit l'application. Dans ce cas, il est donc nécessaire de rajouter des modèles externes traduits exprimés dans le métamodèle de l'utilisateur ou de l'application et obtenus à l'aides de nouveaux traducteurs. Remarquons que cet ajout n'est pas en désaccord avec l'objectif 5. En effet, il n'y est pas précisé que tous les utilisateurs doivent voir les données du SGMB structurées suivant un même métamodèle.

Remarque Notons enfin que, à l'inverse de [Sheth et al.90], nous avons placé la traduction des métamodèles locaux au métamodèle global « au dessus » de l'opération de filtrage consistant à ne considérer qu'une partie des structures des sources de données locales. L'avantage de notre approche est que le SGMB n'a ainsi pas à connaitre l'ensemble des structures des sources de données locales mais uniquement celles qui le concernent.

1.2.2 Architecture du composant transactionnel

Dans la partie 1.2.1, nous avons décrit les différentes étapes permettant de mieux cerner comment les données hétérogènes stockées dans les sources de données locales pouvaient être présentées d'une manière homogène aux utilisateurs à l'aide de diverses transformations. L'architecture ainsi définie est centrée sur les modèles et métamodèles utilisés lors de ces transformations.

Dans cette partie, nous traitons de l'architecture du système transactionnel utilisé pour garantir la cohérence des données du SGMB et donc des sources de données locales. Comme nous l'avons déjà mentionné en introduction, une transaction est un ensemble d'opérations effectuées sur la base de données, garantissant à plusieurs utilisateurs, travaillant simultanément et donc de façon concurrente sur la même base de données, la cohérence de la base de données et de l'exécution de leurs programmes. Nous décrirons plus en détails la notion de transaction et les différentes théories qui s'y rattache dans la partie 1.3. Comme n'importe quel SGBD, un SGMB doit garantir la cohérence des données qu'il stocke virtuellement. Cependant, le système transactionnel du SGMB peut et doit prendre en compte celui des SGBD locaux à travers lesquels les données doivent être accédées puisque le SGMB se comporte comme un utilisateur normal de ces systèmes. L'architecture du composant transactionnel d'un SGMB est donc différente de celle d'un SGBD. Nous en présentons ici les principales caractéristiques.

Dans la suite, nous supposerons que toutes les sources de données locales disposent d'un gestionnaire de transactions. Cette supposition permet de prendre en compte la plupart des SGBD. En ce qui concerne les autres sources de données, telles que les pages WEB ou simplement des fichiers, qui ne disposent pas de gestionnaire transactionnel, des recherches sont en

cours pour permettre leur exploitation à l'aide de systèmes transactionnels mais ne font pas l'objet de ce mémoire. Nous nous concentrerons donc sur les sources de données comportant un système transactionnel.

1.2.2.1 Types de transaction

Un SGMB doit prendre en compte et savoir gérer deux types de transaction afin de respecter d'une part l'objectif 10 et d'autre part, l'autonomie des sources de données locales :

- Les *transactions locales* sont exécutées par la source de données locale hors du contrôle du SGMB. Ces transactions gèrent uniquement les données de la source de données locale sur laquelle elles s'exécutent. Tout utilisateur, en particulier un SGMB, ne peut accéder aux données locales que dans le cadre de l'exécution d'une transaction.
- Les *transaction globales* sont exécutées sous le contrôle du SGMB. Ces transactions gèrent des données situées sur plusieurs sites. Nous appellerons *sous-transaction globale* la projection d'une transaction globale sur chacune des sources de données locales impliquées dans la transaction globale.

Les transactions globales sont gérées au sein du SGMB par un gestionnaire de transaction globale, noté GTM pour Global Transaction Manager en anglais. Le SGMB contient aussi un ensemble de serveurs disposés au dessus des sources de données locales et servant d'interface transactionnelle entre le SGMB et les sources de données locales. Ce gestionnaire et ces serveurs peuvent être modélisés principalement à l'aide de deux architectures respectivement distribuée et centralisée. Nous les présentons ci-dessous.

1.2.2.2 Architecture distribuée

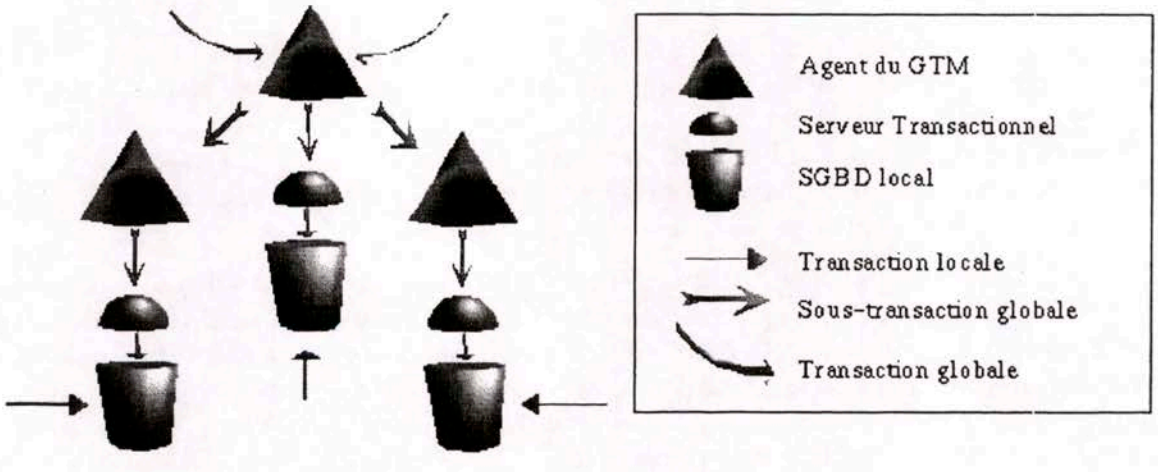


FIG. 1.4 – Architecture distribuée

L'architecture décrite ici distribue sur chaque site local un agent du GTM (voir par exemple [Baldoni et al.95]). Celui-ci est scindé en autant d'agents qu'il y a de sources de données locales. Lorsque, sur un site, un agent du GTM traite une transaction globale, il la décompose en sous-transactions globales devant être exécutées sur les sites concernés par d'autres agents

du GTM. Lorsqu'un agent du GTM traite une sous-transaction globale, il alloue un serveur qui se charge de faire exécuter la transaction locale correspondante par la source de données locale (voir la figure 1.4). La façon précise dont les serveurs et la source de données locale interagissent dépend de l'interface exportée par le SGBD. Une des possibilités est que la source de données locale accepte des opérations individuelles de lecture et d'écriture. Dans ce cas, la marche à suivre pour le serveur est d'ouvrir une transaction locale sur la source de données locale avant toute opération. Puis, lors de chaque lecture ou écriture, il doit attendre respectivement la valeur lue ou la confirmation de l'écriture. Enfin, lorsque le GTM veut valider la transaction globale, c'est-à-dire lorsqu'il veut rendre effectives les modifications apportées à la source de données locale, le serveur envoie une demande de validation à la source de données.

1.2.2.3 Architecture centralisée

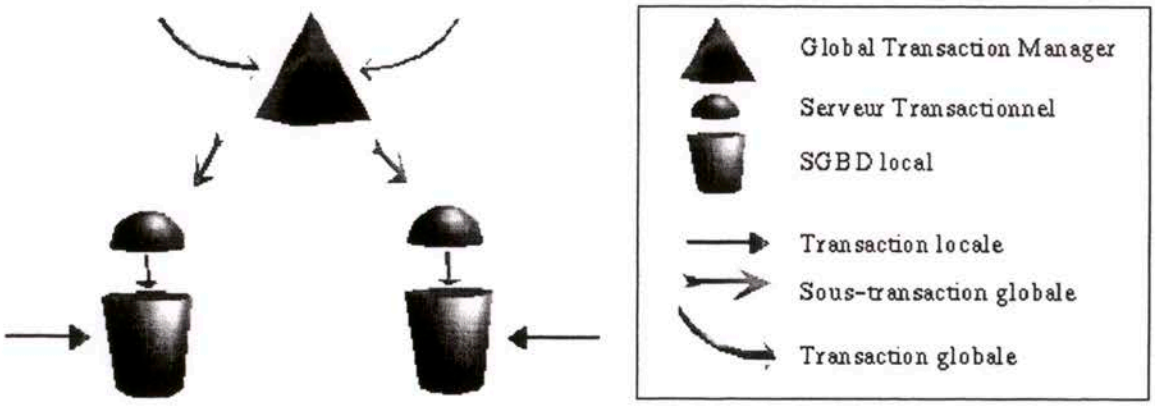


FIG. 1.5 – Architecture centralisée

Cette architecture est un peu plus simple que la précédente, elle repose sur un seul GTM centralisé (voir [Breitbart et al.95] et [Bukhres et al.95]). Celui-ci décompose chaque transaction globale en sous-transactions globales devant être exécutées par le serveur de chacun des sites impliqués dans la transaction globale pour qu'elles soient traitées par le serveur (voir la figure 1.5). Comme précédemment, la façon dont interagissent serveurs et sources de données locales dépend de l'interface exportée par celles-ci.

Le principal inconvénient de cette architecture est que le GTM devient un goulot d'étranglement : toutes les transactions globales doivent transiter par ce composant. Cependant, cette architecture, contrairement à la précédente, a l'avantage de ne pas avoir à dupliquer ni les informations globales permettant de décomposer les transactions globales sur chaque site ni le système réalisant cette décomposition. Toute transaction globale peut ainsi profiter éventuellement d'informations d'optimisation générées lors de l'exécution d'autres transactions globales, ce qui doit permettre l'amélioration des performances du SGMB.

1.2.3 Architecture du gestionnaire de requêtes

Dans la partie 1.2.1, nous avons décrit des processeurs permettant de décomposer des requêtes, de les transformer, et inversement de recomposer et transformer leur résultat. L'architecture

du gestionnaire de requêtes peut ainsi être décrite comme la « projection » du gestionnaire de données, correspondant uniquement au traitement des requêtes. Il est néanmoins possible de définir une architecture du gestionnaire de requête similaire à celle définie pour le gestionnaire transactionnel et correspondant plutôt à une architecture logicielle du gestionnaire permettant de localiser précisément sur le réseau les différents systèmes mis en œuvre pour l'exécution d'une requête.

Dans ce cadre, à l'instar de ce qui a été défini dans la partie 1.2.2, un SGMB doit savoir gérer deux types de requêtes afin de respecter d'une part l'objectif 10 et d'autre part, l'autonomie des sources de données locales :

- Les *requêtes locales* sont exécutées par la source de données locale hors du contrôle du SGMB. Ces requêtes permettent d'exploiter les données de la source de données locale sur laquelle elle s'exécute. Tout utilisateur, en particulier un SGMB, ne peut accéder aux données locales qu'à l'aide de requêtes locales. Notons que les requêtes locales sont exprimées dans le langage de requêtes local.
- Les *requêtes globales* sont exécutées sous le contrôle du SGMB. Ces requêtes doivent permettre d'accéder à des données situées sur plusieurs sites. On appellera *sous-requête globale* la projection d'une requête globale sur chacune des sources de données locales impliquées dans la transaction globale.

Les requêtes globales sont gérées au sein du SGMB par un gestionnaire de requêtes globales, noté GQM pour Global Query Manager en anglais. Le SGMB contient aussi un ensemble de serveurs disposés au dessus des sources de données locales et servant d'interface entre le SGMB et les sources de données locales. Ce gestionnaire et ces serveurs peuvent être modélisés principalement à l'aide de deux architectures, l'une distribuée et l'autre centralisée, similaires à celles décrites en 1.2.2. Notons qu'en règle générale, la même architecture est choisie à la fois pour le gestionnaire transactionnel et le gestionnaire de requêtes pour faciliter l'implantation du SGMB. Enfin, il est possible de faire un lien avec l'architecture de la partie 1.2.1 en répartissant les processeurs à la fois au niveau du GQM, pour ce qui concerne les niveaux externes et conceptuels du SGMB, et au niveau des serveurs, pour ce qui est du niveau interne.

1.3 Gestion transactionnelle dans les SGMB

Comme nous l'avons précédemment mentionné, un SGMB doit, à l'instar de n'importe quel SGBD, garantir la cohérence des données qu'il stocke virtuellement. En 1.2.2, nous avons vu l'architecture générale du gestionnaire transactionnel du SGBD en introduisant les concepts de GTM, de transactions globales et locales, etc ... Dans cette partie, après une introduction des techniques et algorithmes transactionnels utilisés dans les SGBD, nous porterons notre attention sur les problèmes transactionnels induits au sein des SGMB par l'hétérogénéité, la distribution et l'autonomie des sources de données. Puis nous décrirons les différentes solutions proposées dans la littérature pour résoudre ces problèmes.

1.3.1 Techniques transactionnelles dans les SGBD

Dans cette partie, nous considérons qu'une transaction est un ensemble d'opérations de lecture (r pour read) et d'écriture (w pour write) de données terminée par une validation (c pour

commit) ou une annulation (a pour abort)[4]. On dit que deux opérations sur la même donnée entrent en conflit si l'une, au moins, est une écriture. Deux transactions T_i et T_j, s'exécutant en parallèle, sont dites en concurrence si au moins une opération de T_i est en conflit avec une opération de T_j sur une même donnée.

Nous appellerons *histoire transactionnelle* un ensemble d'opérations de lecture et d'écriture appartenant à une ou plusieurs transactions. Une histoire est dite *sérialisée* si elle ne contient que des opérations de transactions consécutives. Une histoire est dite *sérialisable* si les résultats qu'elle produit sont équivalents à ceux d'une histoire sérialisée contenant les mêmes transactions.

Enfin, une histoire transactionnelle est dite *sans conflit* si chaque transaction n'est en concurrence avec aucune autre. Notons qu'une histoire sérialisée est donc sans conflit puisqu'elle ne contient que des transactions consécutives.

1.3.1.1 Propriétés ACID

Afin de garantir la cohérence de la base de données et des applications, les transactions, au sein desquelles se font tous les échanges entre les utilisateurs ou les applications et la base de données, doivent vérifier les propriétés d'Atomicité, de Cohérence, d'Isolation et de Durabilité ou propriétés ACID (voir [Gray et al.93]) :

La propriété d'atomicité. Elle assure que chaque transaction s'exécute soit totalement, et dans ce cas elle est validée, soit pas du tout, et elle est alors annulée. C'est le principe du tout ou rien. Ce principe permet de garantir la cohérence des applications.

La propriété de cohérence. Elle assure que l'exécution d'une transaction laisse la base de données dont l'état avant le début de la transaction était cohérent dans un état cohérent. Cela implique que l'exécution d'une transaction ne doit pas violer les contraintes d'intégrité posées sur la base de données.

La propriété d'isolation. Elle assure que chaque transaction ne voit pas les effets d'une transaction non validée. Tout se passe comme si chaque transaction s'exécutait sur une base de données indépendante. Remarquons que si des transactions s'exécutant en parallèle ne sont pas concurrentes, la propriété d'isolation est respectée. Une histoire sérialisée de même qu'une histoire sérialisable respecte donc la propriété d'isolation.

La propriété de durabilité. Elle assure que les effets d'une transaction validée sont persistants. En d'autres termes, les modifications apportées par une transaction validée sont enregistrées dans la base de données.

[4]Il est aussi possible de décomposer une transaction en opérations plus complexes, ce qui implique de définir des algorithmes différents de ceux présentés dans ce mémoire. Nous nous restreindrons néanmoins à cette description car c'est d'une part la plus générale, et d'autre part, la plus répandue et utilisée dans les systèmes actuels.

Dans un système centralisé, les propriétés d'atomicité et de durabilité sont garanties par des algorithmes de journalisation et de reprise sur pannes. La propriété de cohérence peut soit être garantie par le système qui alors annule les transactions ne vérifiant pas cette propriété, soit être laissée à la charge de l'utilisateur. Nous étudions les algorithmes et techniques permettant de garantir la propriété d'isolation dans les parties suivantes. Enfin, nous verrons en 1.3.1.3 les problèmes liés à la répartition des données du SGBD et donc au passage à un système distribué. Nous y détaillerons aussi les problèmes liés à l'imbrication des transactions.

1.3.1.2 Algorithmes garantissant l'isolation des transactions

Nous présentons ci-dessous les algorithmes les plus connus dont ceux utilisés dans les SGBD actuels. Ils ont été décrits en autres dans [Bernstein et al.87], [Kung et al.81] et [Kaiser93].

Verrouillage à 2 phases. Cet algorithme repose sur le fait surprenant qu'il n'y ait que trois formes simples de violation de l'isolation lorsque l'on considère les transactions comme des ensembles de lectures et écritures de données. Elles sont la base des notions de conflit entre données et de concurrence entre transactions :

- Une *lecture incohérente* (LI) se produit dès lors qu'une transaction T_i lit une donnée entre la lecture et l'écriture de cette même donnée par une autre transaction T_j. La donnée lue par T_i a donc une valeur non cohérente car ce n'est pas la valeur finale de cette donnée telle que produite par T_j.
- Une *lecture non reproductible* (LNR) se produit dès lors qu'une transaction T_i écrit une donnée entre deux lectures de cette même donnée par une autre transaction T_j. Les deux valeurs lues par T_j sont donc incohérentes car différentes.
- Une *perte de mise à jour* (PMJ) se produit lorsqu'une transaction T_i écrit une donnée entre la lecture ou l'écriture et l'écriture de cette même donnée par une autre transaction T_j. La mise à jour effectuée par T_i n'est donc pas prise en compte par T_j et est ainsi perdue.

Ceci nous montre que deux transactions concurrentes sont isolées si et seulement si l'historique de ces deux transactions ne comporte aucune des quatre séquences d'opérations suivantes, $r_k[x]$ étant la lecture de la donnée x par la transaction k et $w_k[x]$ étant une écriture de la donnée x par la transaction k :

$$\begin{aligned} LI &: \quad r_j[x]r_i[x]w_j[x] \\ LNR &: \quad r_j[x]w_i[x]r_j[x] \\ PMJ &: \quad \begin{array}{l} r_j[x]w_i[x]w_j[x] \\ w_j[x]w_i[x]w_j[x] \end{array} \end{aligned} \tag{1.1}$$

Pour éviter que les séquences précédentes ne se produisent, il suffit d'empêcher certaines opérations de se réaliser en posant des verrous sur les données. L'algorithme de verrouillage suit les règles suivantes.

- Avant chaque lecture ou écriture d'une donnée, la transaction pose respectivement un verrou en lecture ou un verrou en écriture sur cette donnée.

- Il est possible de poser un verrou en lecture sur une donnée si celle-ci ne possède pas déjà un verrou en écriture.
- Il est possible de poser un verrou en écriture sur une donnée si celle-ci ne possède pas déjà ni un verrou en lecture, ni un verrou en écriture.
- Enfin, dès qu'une transaction a relâchée un verrou, il ne lui est plus possible d'acquérir de nouveaux verrous.

L'algorithme se déroule donc en deux phases distinctes, une première phase d'acquisition des verrous (growing phase) et une seconde où les verrous sont relâchés (shrinking phase). En pratique, les verrous sont tous relâchés en fin de transaction. Cette variante de l'algorithme se nomme verrouillage à 2 phases strict. Bernstein montre théoriquement dans [Bernstein et al.87] que l'exécution de l'algorithme de verrouillage à 2 phases produit des histoires sérialisables. Il est néanmoins facile de voir que cet algorithme garantit que les quatre séquences d'opération présentées ci-dessus ne peuvent se produire.

Le principal défaut de cet algorithme est le suivant. Lorsqu'une transaction veut accéder à une grande partie de la base, par exemple à une table entière en relationnel, elle doit poser autant de verrous que de données et donc faire baisser les performances de tout le système. Pour résoudre ce problème, d'autres algorithmes de verrouillage ont été étudié, notamment en utilisant la structure hiérarchique de la plupart des bases de données.

Enfin, tous les algorithmes de verrouillages connus peuvent provoquer des interblocages ou verrous mortels. Par exemple, si une transaction T_i attend la levée d'un verrou d'une transaction T_j qui attend la levée d'un verrou d'une transaction T_k qui attend la levée d'un verrou posé par T_i, alors l'algorithme attend indéfiniment induisant un verrou mortel. Deux approches peuvent être choisies pour résoudre ce problème. La première est une solution curative. Il s'agit de détecter les verrous mortels formés puis de les casser en annulant l'une des transactions y participant. L'algorithme le plus utilisé mettant en œuvre cette approche construit le graphe des attentes inter-transactions, appelé Wait-For-Graph, pour en détecter les circuits, ceux-ci correspondant à des verrous mortels. La deuxième solution est préventive. Il s'agit d'imposer un délai de garde pour l'exécution de chaque opération d'une transaction au delà duquel la transaction concernée est annulée.

Estampillage L'algorithme présenté dans la partie précédente est majoritairement utilisé dans les SGBD actuels. Cependant, il présente un défaut important qui impose de mettre en œuvre des algorithmes de résolution et/ou de détection de verrous mortels qui grèvent les performances du système. Pour pallier ce défaut, il est possible d'utiliser un autre type d'algorithme pour garantir la sérialisabilité des histoires transactionnelles (voir par exemple, [Madria97]).

Celui-ci associe à chaque transaction T_i, ainsi qu'à chacune de ses opérations, une estampille unique notée ts_i. Une estampille peut par exemple être la valeur d'un compteur incrémenté pour chaque nouvelle transaction. Les opérations conflictuelles sont alors ordonnées suivant leurs estampilles respectives en fonction de la règle énoncée ci-dessous :

- Si op_i et op_j sont deux opérations en conflit alors op_i sera exécutée avant op_j si et seulement si $ts_i < ts_j$.

En respectant cette règle, on s'assure que deux opérations conflictuelles sont toujours exécutées dans l'ordre de leur estampille. En conséquence, l'exécution selon l'ordre ainsi défini est

équivalent à l'exécution en série des transactions ordonnées suivant leurs estampilles. Cet algorithme génère donc automatiquement des histoires transactionnelles sérialisables. Le défaut de cet algorithme est le risque de générer plus d'annulations que l'algorithme par verrouillage.

Gestion par versions de données Jusqu'à présent, nous avons considéré que toutes les transactions travaillaient sur les données dans un mode dit de mises à jour immédiates. Ce mode correspond à l'enregistrement immédiat des modifications effectuées par les transactions. Il est possible de définir un autre mode dans lequel chaque transaction s'exécute dans un environnement qui lui est propre, c'est-à-dire distinct de l'environnement des autres transactions. En d'autres termes, dans ce mode, lorsqu'une transaction veut effectuer une modification, une copie de la donnée concernée est créée dans l'environnement de la transaction. Seule cette copie peut-être modifiée lors de l'exécution de la transaction. La donnée initiale n'est, quant à elle, mise à jour que lorsque la transaction est validée. Ce mode est appelé mode à mises à jour différées. Notons que dans ce mode, la propriété d'atomicité est automatiquement vérifiée puisque si la transaction est abandonnée, aucune mise à jour n'est effectuée sur la base de données.

L'algorithme de contrôle de concurrence par verrouillage, décrit précédemment, a été développé initialement pour le mode de mises à jour immédiates. Il peut néanmoins être modifié pour fonctionner en mises à jour différées. Il est cependant plus intéressant d'utiliser un algorithme plus adapté au mode de mises à jour différées tel que celui décrit ci-après.

Dans le mode de mises à jour différées, chaque transaction est exécutée dans un environnement qui lui est propre. Par conséquent, il est tentant de vouloir laisser s'installer des dépendances entre les transactions en cours d'exécution car, dans ce mode, les dépendances ne risquent pas d'entraîner d'incohérence entre les données tant que les transactions n'ont pas été validées. L'algorithme que nous présentons ici est dit optimiste et n'est mis en jeu que lors de la validation des transactions, à l'opposé des algorithmes précédemment décrits, dits pessimistes ou continus, qui sont mis en œuvre pendant l'intégralité de l'exécution des transactions.

L'algorithme le plus simple pour garantir l'isolation des transactions en mode de mises à jour différées est appelé algorithme de certification par ensemble d'objets. Il consiste à insérer une phase de certification entre la fin de l'exécution de chaque transaction et leur validation. Lors de la certification d'une transaction T, on contrôle que tous ses conflits avec les transactions T_i précédemment validées n'impliquent pas une perte de l'isolation des transactions. Par rapport aux conflits décrits par les séquences (1.1), cela revient à vérifier que l'historique de chaque paire (T_i, T) ne comporte pas la séquence d'opérations suivante sur une même donnée, $r_k[x]$ étant la lecture de la donnée x par la transaction k et $w_k[x]$ étant une écriture de la donnée x par la transaction k :

$$r_i[x]\, r[x]\, w_i[x] \tag{1.2}$$

Notons que les autres séquences n'entraînent pas de conflits en mode différé. Pour réaliser ce contrôle, on maintient, pour chaque transaction T en cours d'exécution les ensembles notés $L(T)$ et $E(T)$, des données respectivement lues et écrites par la transaction. Lors de la certification de T, l'existence d'un conflit avec une transaction validée T_i est mise en évidence par la condition suivante :

$$E(T_i) \cap L(T) \neq \emptyset \tag{1.3}$$

Si la certification de T montre l'existence d'un conflit, T est annulée, sinon elle est validée.

Le défaut de cet algorithme est qu'il induit souvent de nombreuses annulations. Certains travaux tels que ceux présentés en [Llirbat et al.96] cherchent à résoudre ce problème.

1.3.1.3 Transactions Evoluées

Pour terminer cette rapide description des techniques transactionnelles, nous présentons dans cette partie les conséquences de l'imbrication ou de la distribution des transactions sur les algorithmes de contrôle de concurrence ou de garantie de l'atomicité (voir [Barghouti et al.91]).

Transactions emboîtées Dans le modèle transactionnel pris en compte jusqu'à présent, dit modèle transactionnel plat, les transactions toutes entières constituent des unités d'abandon et de reprise, ceci pour vérifier la propriété d'atomicité. Il en résulte que toutes les modifications apportées par les transactions sont perdues en cas d'annulation. Cette propriété d'atomicité a de lourdes conséquences lorsque l'on souhaite mettre en œuvre des transactions de longue durée et/ou des traitements répartis sur plusieurs sites. Pour résoudre ce problème, il est souhaitable de mettre en œuvre de nouveaux modèles transactionnels permettant d'annuler des transactions par partie. De plus, bien que le modèle transactionnel plat n'interdise pas à un programmeur d'exprimer des traitements parallèles locaux à une même transaction, il n'en reste pas moins que le contrôle de concurrence reste à sa charge. Il est donc là de plus nécessaire d'utiliser de nouveaux modèles transactionnels garantissant un contrôle de concurrence même lors de traitements parallèles inter-transactions.

Le modèle des transactions emboîtées introduit par [Moss85] consiste à permettre l'imbrication de transactions au sein d'autres transactions. Ainsi, une transaction n'est plus une entité monolithique comme cela était le cas dans le modèle des transactions plates, mais peut contenir un nombre quelconque de sous-transactions, qui à leur tour peuvent être composées de sous-transactions et ainsi de suite. Un ensemble de transactions définies suivant le modèle des transactions emboîtées peut ainsi être représentée par une forêt[5].

Pour garantir les propriétés ACID des transactions racines, entités globalement équivalentes à des transactions plates, la relation d'emboîtement induit les règles de fonctionnement suivantes :

- Une transaction T_j, fille d'une (sous-)transaction T_i démarre après T_i et se termine avant elle.
- L'abandon d'une sous-transaction entraîne l'abandon de ses sous-transactions.
- La validation d'une sous-transaction est conditionnée par la validation de sa (sous-)transaction mère. En d'autres termes, ce n'est qu'à la validation de la transaction racine que les mises à jour effectuées par ses sous-transactions validées deviennent durables.

Notons que ces règles impliquent que chaque sous-transaction ne vérifie que les propriétés AI. Diverses variantes de ce modèle ont été proposées dans la littérature, leurs différences

[5] Une forêt est un ensemble d'arbres !

provenant principalement de la façon dont sont traitées les possibilités de concurrence entre une transaction mère et ses sous-transactions ou entre des transactions sœurs.

Pour terminer sur ce point, il est intéressant de discuter des conséquences de l'emboîtement des transactions sur les algorithmes de contrôle de concurrence et en particulier sur le mécanisme de verrouillage à 2 phases. Plaçons nous dans un contexte où seules les sous-transactions feuilles accèdent à des objets partagés, au moyen d'opérations lire et écrire. Dans ce cadre, [Moss85] a défini une extension de l'algorithme de verrouillage à 2 phases appelé verrouillage emboîté à deux phases. Comme l'algorithme défini précédemment, celui-ci impose que toute opération sur une donnée soit précédée de la pose d'un verrou ad-hoc sur cette donnée. L'algorithme de verrouillage emboîté suit les règles suivantes :

- Avant chaque lecture ou écriture d'une donnée, une sous-transaction doit poser respectivement un verrou en lecture ou un verrou en écriture sur cette même donnée.
- Il est possible de poser un verrou en lecture sur une donnée si celle-ci ne possède pas déjà de verrou en écriture ou si toutes les transactions qui possèdent un verrou en écriture sur la donnée sont des ancêtres de la sous-transaction.
- Il est possible de poser un verrou en écriture sur une donnée si celle-ci ne possède pas déjà ni de verrou en lecture, ni de verrou en écriture ou si toutes les transactions qui possèdent un verrou en écriture ou en lecture sur la donnée sont des ancêtres de la sous-transaction.
- Lorsqu'une sous-transaction est validée, tous ses verrous sont anti-hérités par sa transaction mère. En d'autres termes, sa transaction mère acquiert ces verrous par héritage inverse.
- Lorsqu'une sous-transaction est annulée, tous ses verrous sont relâchés.

Il est assez facile de voir que cet algorithme garantit l'isolation des transactions du fait que seules les sous-transactions filles accèdent aux données. Notons que cet algorithme peut produire des verrous mortels de détection plus complexe que ceux produits par l'algorithme simple. Le lecteur pourra se référer à [Rukoz91] pour avoir un aperçu des techniques employées pour la détection des verrous mortels induits par l'algorithme emboîté.

Enfin, il est possible d'étendre l'algorithme présenté ici pour permettre une certaine concurrence entre transactions d'une même hiérarchie (voir [Daynes et al.95] ou [Harder et al.93])

Transactions distribuées Pour terminer cette présentation des systèmes transactionnels, intéressons nous aux systèmes transactionnels distribués ou répartis ce qui nous rapprochera des SGMB qui sont une généralisation de tels systèmes. Plusieurs motivations peuvent conduire à distribuer une application sur différents sites. Il s'agit par exemple de mettre en œuvre des applications de télécommunication qui sont intrinsèquement réparties, ou bien des applications critiques en temps de réponse telles que les applications de contrôle du trafic aérien qui nécessitent d'avoir accès à plusieurs sites simultanément en cas de défaillance d'un site, ou encore des applications dont les traitements sont distribués sur plusieurs sites pour en améliorer les performances d'exécution. Dans la suite de cette partie, nous passons en revue les différentes propriétés que doivent vérifier les transactions pour déterminer les problèmes posés par une architecture répartie sur les systèmes transactionnels.

Il est assez simple de voir que la propriété de durabilité des transactions réparties est équivalente à garantir cette propriété sur chaque site local. En effet, une fois que les données sont sauvegardées localement et donc durables localement, elles le sont aussi globalement.

Dans le cas de la propriété d'isolation, les mécanismes de contrôle de concurrence développés dans un contexte centralisé ne se transposent pas toujours dans un système réparti. En effet, nous avons vu que les algorithmes de contrôle de concurrence nécessitent au minimum la connaissance de l'état de toutes les données impliquées dans une transaction lors de sa validation. Or, dans un contexte réparti, cet état est distribué sur chaque site. Il est donc nécessaire, dans un tel contexte, que chaque site rende disponible globalement l'état des données mises en œuvre dans une transaction globale et que ces états soit cohérents entre les différents sites pour pouvoir être traités globalement. En particulier, si chaque site n'utilise pas le même contrôle de concurrence, il leur sera difficile de fournir des états susceptibles de servir lors de la mise en œuvre d'un contrôle de concurrence global. En pratique, les systèmes répartis dits homogènes utilisent donc le même contrôle de concurrence sur chaque site et peuvent soit élire un site responsable pour traiter globalement le contrôle de concurrence ce qui implique de lui fournir les informations nécessaires, soit adapter les algorithmes précédemment décrits pour répartir leur traitement global sur chacun des sites.

Enfin, pour assurer la propriété d'atomicité, les mêmes mécanismes de journalisation et de reprise utilisés dans un contexte centralisé peuvent être utilisés tels quels dans un contexte réparti. Le problème supplémentaire qui se pose ici est d'assurer que tous les sites impliqués dans une transaction globale prennent la même décision concernant la validation de celle-ci. En effet, supposons qu'une transaction consiste à effectuer un virement d'un compte stocké sur un site sur un autre compte situé sur un site différent. Cette transaction doit effectuer un débit sur un compte et un crédit sur l'autre. Il est nécessaire qu'à la fois le débit et le crédit soient effectués pour garantir l'atomicité de la transaction. Il faut donc que la transaction soit validée à la fois sur les deux sites ou sur aucun d'entre eux. Pour garantir l'atomicité dans un contexte réparti, il est courant d'utiliser le protocole de validation en 2 phases [Gray et al.93]. Pour prendre une image, ce protocole est similaire à celui utilisé par exemple dans la cérémonie du mariage où un coordinateur (le maire, le prêtre, le rabin, l'imam...) demande à chacun des futurs époux s'ils sont d'accord pour se marier. Si les deux sont d'accord, c'est-à-dire, si les deux répondent oui, le mariage est décidé. Sinon, c'est-à-dire si au moins l'un des deux répond non, le mariage est annulé. Le protocole de validation en deux phases est similaire et peut être décrits par les étapes suivantes

- Préparation : le coordinateur, par exemple le site initiateur de la transaction globale, demande à tous les site participants de voter en leur envoyant un message `préparer`.
- Vote : chaque participant qui reçoit ce message retourne son vote au coordinateur. Si le participant ne reçoit pas ce message alors que la transaction est censée être terminée, il peut suspecter une défaillance du coordinateur et renvoyer non.
- Décision du coordinateur : Si l'un des votes reçus par le coordinateur est non ou s'il suspecte la défaillance d'un site (par exemple s'il ne reçoit pas de réponse au delà d'un délai de garde), le coordinateur décide d'annuler la transaction, sinon il décide de la valider.
- Décision des participants : Lorsque le coordinateur à décidé, chaque participant doit suivre cette décision et soit annuler soit valider la transaction.

Ce protocole se nomme protocole de validation en deux phases car il y a une phase de préparation avant la phase de décision et d'exécution de cette décision. Il existe différentes variantes de ce protocole. Par exemple, pour des raisons d'efficacité, les sites participants peuvent valider leur part de la transaction sans attendre la décision du site coordinateur. Si le coordinateur décide d'annuler, il faut alors exécuter une transaction de compensation permettant de rendre la base cohérente. Cette technique s'appelle protocole de validation en 2 phases optimiste et

repose sur le fait que pratiquement plus de 97% des transactions sont validées.

Une des limitations majeures du 2PC est la possibilité de blocage lorsque le coordinateur est défaillant (il n'est pas toujours possible aux participants de déterminer que c'est le cas). Pour résoudre ce problème il est possible d'utiliser alors un protocole de validation en 3 phases comme par exemple celui présenté en [Skeen81] qui rajoute une phase où le coordinateur diffuse son intention de décider et s'assure qu'elle est connue par tous les participants. Cette phase à pour but de permettre aux participants d'être toujours en mesure de se concerter en cas de défaillance du coordinateur.

1.3.2 Gestion des Transactions dans les SGMB

Dans les parties précédentes, nous avons vu différentes techniques permettant de garantir les propriétés ACID des transactions au sein des SGBD. Nous nous intéressons dans cette partie plus particulièrement aux problèmes soulevés par la gestion des transactions au sein de SGMB. Un SGMB étant avant tout un SGBD distribué, les techniques présentées précédemment devraient pouvoir s'adapter aux SGMB. Cependant, le respect des principes d'autonomie des sources de données locales induit de nouveaux problèmes transactionnels. Nous les détaillons ci-dessous.

1.3.2.1 Le Problème de sérialisabilité globale

Nous avons vu précédemment que garantir la sérialisabilité d'une histoire transactionnelle permet de respecter la propriété d'isolation des transactions.

Comme nous l'avons présenté en 1.3.1.3, les différentes solutions garantissant une sérialisabilité globale des transactions dans un système distribué homogène, suppose que tous les sites locaux impliqués dans le SGBD distribué partagent leurs informations de contrôle et utilisent le même type d'algorithme de gestion de la concurrence (verrouillage à 2 phases, estampillage, ...).

Dans un environnement hétérogène ces conditions ne sont pas vérifiées. Les sources de données locales peuvent utiliser des algorithmes de contrôle de concurrence divers et, pour respecter leurs autonomies de communication, ne partagent aucune information de contrôle sur l'exécution de leurs transactions. Notamment, le GTM n'a pas connaissance des transactions locales s'exécutant directement au dessus des sources de données locales. Dans un tel environnement, même une exécution non-conflictuelle de transactions globales ne suffit pas à assurer la sérialisabilité des histoires globales. Pour illustrer ce fait, considérons l'exemple suivant.

Soit un SGMB constitué de deux sources de données locales s_1 et s_2. Soient a et b deux données de s_1 et c et d deux données de s_2. Nous noterons $r_k[x]$ une lecture de la donnée x par la transaction k et $w_k[x]$ une écriture de la donnée x par la transaction k. Enfin, nous noterons c_k la validation de la transaction k.

Considérons les deux transactions globales T_1 et T_2 telles que :

$$\begin{array}{rcl} T_1 & = & \{r_1[a], r_1[c]\} \\ T_2 & = & \{r_2[b], r_2[d]\} \end{array} \tag{1.4}$$

De plus considérons les deux transactions locales suivantes T_3 et T_4 s'exécutant respectivement sur s_1 et sur s_2, et ce de manière concurrente aux transactions T_1 et T_2 :

$$\begin{array}{rcl} T_3 & = & \{w_3[a], w_3[b]\} \\ T_4 & = & \{w_4[c], w_4[d]\} \end{array} \tag{1.5}$$

Supposons que T_1 s'exécute et valide, suivi de l'exécution et de la validation de T_2. Sur chacun des sites locaux, supposés produire des histoires sérialisables, il est possible que les deux histoires suivantes soient exécutées :

$$s_1 : r_1[a]c_1w_3[a]w_3[b]c_3r_2[b]c_2 \tag{1.6}$$

$$s_2 : w_4[c]r_1[c]c_1r_2[d]c_2w_4[d]c_4 \tag{1.7}$$

Ces deux histoires sont en effet sérialisables car l'exécution de la première est équivalente à l'exécution en série de T_1, T_3 et T_2, et l'exécution de la seconde est équivalente à l'exécution en série de T_2, T_4 et T_1. Néanmoins, on remarque qu'il n'y a pas sérialisabilité globale car sur s_1 T_1 précède T_2 alors que sur s_2 c'est l'inverse. La sérialisabilité locale ne suffit donc pas à assurer la sérialisabilité globale même dans le cas d'exécutions globales sérielles comme le montre Breitbart dans [Breitbart et al.95]. Le problème est causé par les transactions locales qui créent un conflit indirect entre les transactions globales. Comme le GTM, pour respecter l'autonomie de communication, n'a pas connaissance des transactions locales, il ne peut connaître les conflits indirects éventuels et donc résoudre classiquement le problème.

Pour essayer de le résoudre, plusieurs cas de figure sont à prendre en compte. Premièrement, si toutes les sources de données locales utilisent un algorithme d'isolation strict comme par exemple le verrouillage à 2 phases strict, il suffit que le GTM n'envoie l'ordre de valider aux sources locales que lorsque toutes les opérations des sous-transactions globales sont terminées pour que la sérialisabilité globale soit respectée [Mehrotra93]. Ceci est facile à voir sur l'exemple précédent si l'on suppose que les sources de données locales utilisent un verrouillage à 2 phases strict, l'ordonnancement choisi n'est alors plus possible. L'inconvénient majeur de cette méthode est que les sources de données locales doivent conserver des verrous jusqu'à la validation sur tous les sites des sous-transactions globales, ceci entraînant une augmentation des conflits éventuels.

Pour résoudre le problème dans des cas plus généraux où l'on ne fait aucune hypothèse sur l'algorithme d'isolation utilisé par les sources de données locales, il existe principalement deux approches. La première part du principe qu'il est possible de créer artificiellement des conflits directs globaux entre les transactions globales forçant ainsi les transactions à s'exécuter de façon globalement sérialisable (voir [Georgakopoulos91]). La seconde approche consiste à utiliser un critère d'isolation différent de la sérialisabilité globale (voir [Mehrotra91]). En effet, nous avons vu que la sérialisabilité était suffisante pour assurer l'isolation des transactions mais pas nécessaire, d'autres critères d'isolation peuvent ainsi être plus adaptés aux SGMB. Nous détaillons ci-dessous ces deux approches.

La méthode des Tickets Une première façon de résoudre le problème de sérialisabilité globale lorsqu'on suppose uniquement que les SGBD locaux garantissent des exécutions locales sérialisables et sans interblocage est de forcer toutes les transactions globales concurrentes à entrer en conflit. Nous avons vu en effet, que lorsque ce n'est pas le cas, un conflit indirect peut-être induit par des transactions locales et rendre les transactions globales non sérialisables globalement, ce qui n'est pas détectable par le GTM. Forcer un conflit direct entre chaque transaction globale est alors un moyen de faire en sorte que le GTM puisse détecter la non-sérialisabilité des transactions et ainsi garantir l'isolation des transactions globales. Pour ce faire, il est possible d'utiliser la méthode dite des tickets. Un ticket est une donnée particulière créée sur chaque site pris en compte dans le SGMB. Nous appellerons $ticket_k$ le ticket associé au site s_k. Le gestionnaire de transaction force chaque transaction globale à lire puis incrémenter le ticket associé à chaque site accédé. La valeur lue de $ticket_k$ par une transaction globale indique son ordre de sérialisation sur le site s_k. Il est donc possible, à l'aide de cet ordre de construire un graphe de sérialisation global dans lequel les transactions sont les nœuds et les conflits entre transactions sont les arêtes. Si un circuit apparaît dans le graphe, l'isolation n'est pas respectée et une des transactions participant au circuit doit être annulée.

Reprenons l'exemple précédent en appliquant cette technique. Tout d'abord, comme le GTM force chaque transaction à lire puis incrémenter le ticket sur chaque site (nous supposerons que la valeur initiale de chaque ticket est 0), les deux transactions globales sont alors composées de la façon suivante :

$$\begin{aligned} T_1 &= \{r_1[ticket_1], w_1[ticket_1], r_1[a], r_1[ticket_2], w_1[ticket_2], r_1[c]\} \\ T_2 &= \{r_2[ticket_1], w_2[ticket_1], r_2[b], r_2[ticket_2], w_2[ticket_2], r_2[d]\} \end{aligned} \tag{1.8}$$

Supposons que T_1 accède aux sites s_1 et s_2 avant T_2 (ou inversement), les deux transactions globales sont alors globalement sérialisable car il n'y a pas de circuit dans le graphe associé à l'ordre des tickets. Cependant, si la transaction T_4 génère un conflit indirect sur s_2, il y a alors non sérialisabilité des transactions sur s_2à cause de l'écriture du ticket ce qui abandonne l'une des transactions incriminée.

Supposons que T_1 et T_2 n'accèdent pas dans le même ordre aux sites s_1 et s_2 (elles sont donc non sérialisables globalement). Dans ce cas, même si l'exécution de T_1 et T_2 est sérialisable sur chaque site, comme le graphe global associé aux tickets contient un circuit, l'une des deux transactions sera abandonnée par le GTM.

D'autres critères d'isolation La méthode précédente résoud le problème dans le cas général. Cependant, elle impose de rendre conflictuelles des transactions qui ne l'étaient pas forcément, ce qui peut impliquer une diminution des performances du système. Une autre solution est de chercher d'autres critères d'isolation que la sérialisabilité. Nous avons vu en effet que la sérialisabilité était un critère suffisant pour garantir l'isolation des transactions mais pas nécessaire. Il est alors possible que des transactions non sérialisables soient tout de même isolées.

La propriété d'isolation peut être considérée comme une propriété permettant de garantir la cohérence de la base de données en cas d'accès concurrents. Lorsque sont posées des contraintes d'intégrité sur une base de données (par exemple la contrainte spécifiant que le solde d'un compte doit toujours être positif), si l'on suppose que les transactions s'exécutent en série, il est

très facile de garantir la cohérence de la base (cela incombe souvent du reste au programmeur). Cependant, si l'on admet que plusieurs transactions peuvent s'exécuter simultanément, il est possible que même si l'exécution seule de chaque transaction conserve la cohérence, leur exécution en parallèle induise des incohérences d'où la nécessité d'imposer un contrôle de l'isolation des transactions.

Dans un SGMB, il existe deux types de contraintes d'intégrités : les contraintes d'intégrités locales, définies sur les données locales et dont la vérification incombe à chaque site et les contraintes d'intégrités globales posées sur les données gérées par le SGMB et dont la vérification incombe au SGMB. Si l'on impose qu'il n'est possible de définir que des contraintes d'intégrités locales, il est montré dans [Mehrotra91] que l'exécution de transactions globales est isolée.

D'autres recherches ont été menées pour assouplir cette règle (voir [Mehrotra93]). Dans ce cas, il est possible de poser des contraintes d'intégrité globales à la condition d'interdire aux transactions locales d'accéder aux données manipulées au sein de transactions globales et sur lesquelles portent des contraintes d'intégrité globales. En imposant cette règle, l'exécution des transactions globales est là aussi isolée au sens où elle préserve les contraintes d'intégrité.

1.3.2.2 Le problème d'Atomicité globale

Dans cette partie, nous considérons les problèmes induits par d'éventuels échecs de validation des transactions. Un SGMB, comme tout SGBD, se doit de respecter l'atomicité des transactions globales. Comme nous l'avons dejà mentionné, chaque transaction globale est décomposée en sous-transactions globales envoyées sur chaque source de données locale. En supposant que ces dernières garantissent l'atomicité des transactions qu'elles traitent, le rôle du GTM est réduit à garantir soit la validation de toutes les sous-transactions globales sur chaque source de données locale, soit leur annulation sur chaque site.

Nous avons présenté en 1.3.1.3 l'algorithme de validation à 2 phases qui garantit cette propriété dans le cadre de SGBD distribués. Cependant, cette approche ne convient pas complètement pour les SGMB. Tout d'abord, chaque source de données locale peut ne pas disposer de l'opération de préparation nécessaire au bon déroulement de l'algorithme en 2 phases. De plus, même dans le cas ou l'opération de préparation est disponible, celle-ci implique une obéissance des sources de données locales envers le GTM. En effet, l'opération de préparation signifie que lorsque le GTM reçoit l'accord de préparation, il peut alors envoyer un ordre de validation qui doit être exécuté par chaque source de données locale. Or, ceci est contraire à la propriété d'autonomie d'exécution des sources de données locales, celles-ci devant rester maître de la décision de valider ou d'annuler.

Si l'on ne souhaite pas respecter l'autonomie d'exécution des sources de données locales, il est possible d'utiliser la méthode 2PC Agent[6] (voir [Wolski et al.90]) pour garantir l'atomicité des transactions globales. Cette approche simule le protocole de commit à 2 phases à l'aide d'agents, intermédiaires entre la source de données locale et le GTM. Chaque agent exporte l'opération de préparation et impose qu'aucune transaction locale ne modifie les données manipulées par les transactions globales. Dans ce cadre très restrictif, cette méthode garantit l'atomicité des transactions globales.

[6]Ici, agent est à prendre au sens fonctionnel du terme et non au sens des formalismes à objets ou de l'intelligence artificielle. Ici, un agent est simplement un module ou un processus autonome.

Dans le cas le plus général, on peut utiliser plusieurs variantes de la technique précédente (voir [Breitbart et al.95]). Ces approches se basent comme précédemment sur des agents exportant l'opération de préparation. Cependant, en vertu de l'autonomie d'exécution, les sources de données locales peuvent, cette fois, annuler unilatéralement les transactions même après la phase de préparation. Dans ce cas, pour garantir l'atomicité, il est alors nécessaire de réaliser l'une des opérations présentées ci-dessous.

Redo. Cette alternative consiste à soumettre une nouvelle transaction contenant toutes les écritures de la sous-transaction globale précédemment annulée. Cette méthode nécessite l'archivage de toutes les mises à jour que tentent les sous-transactions globales. Si la nouvelle transaction échoue à nouveau, elle est soumise jusqu'à sa validation. L'inconvénient majeur de cette méthode est qu'elle peut conduire, dans certains cas, à une perte de l'isolation des transactions

Retry. Pour pallier le défaut précédent, une deuxième solution est de soumettre à nouveau et entièrement la sous-transaction globale annulée. Notons que pour garantir l'isolation des transactions et donc la cohérence de leur exécution, cette approche nécessite qu'il n'y ait pas de conflits entre sous-transactions d'une même transaction globale. De plus, il est nécessaire qu'au bout d'un certain nombre d'essais, la sous-transaction concernée puisse être validée[7].

Compensate. Enfin, une troisième approche peut-être envisagée. Il s'agit non plus de réessayer les opérations unilatéralement annulées mais d'annuler les sous-transactions déjà validées. Pour ce faire, une nouvelle transaction, dite de compensation, est soumise pour chaque sous-transaction déjà validée. Chaque transaction de compensation contient les opérations nécessaires pour annuler les effets de la sous-transaction globales déjà validée correspondante. Cette approche a aussi ses inconvénients. En effet, toutes les transactions ne peuvent être compensées. Comment, par exemple, compenser l'envoi d'un missile ou tout simplement d'un courrier ? De plus, d'autres transactions locales peuvent avoir eu connaissance des effets de la sous-transaction globale avant que celle-ci ne soit compensée, ceci impliquant une perte de l'isolation des transactions.

1.3.2.3 Le Problème des verrous mortels

Nous avons vu précédemment que les algorithmes de verrouillage garantissant l'isolation des transactions pouvaient induire des verrous mortels entre plusieurs transactions. En supposant que toutes les sources de données locales ont un mécanisme de détection et/ou de suppression des verrous mortels, il est possible que l'exécution de transactions globales même non conflictuelles induise des verrous mortels globaux comme le montre l'exemple suivant.

Reprenons les notations de la partie 1.3.2.1. Soit un SGMB constitué de deux sources de données locales s_1 et s_2. Soient a et b deux données de s_1, et soient c et d deux données de s_2. Nous noterons $r_i[x]$, respectivement $w_i[x]$, une lecture, respectivement une écriture, de la

[7] Ce n'est pas forcément le cas. Par exemple, si l'objet d'une transaction est d'effectuer le débit d'un compte vide !

donnée x par la transaction T_i. Enfin, nous noterons c_i la validation de la transaction T_i. Nous supposerons que les sources de données locales utilisent un verrouillage à deux phases.

Considérons les deux transactions globales T_1 et T_2 telles que :

$$\begin{array}{rcl} T_1 & = & \{r_1[a], r_1[d]\} \\ T_2 & = & \{r_2[c], r_2[d]\} \end{array} \tag{1.9}$$

De plus considérons les deux transactions locales suivantes T_3 et T_4 s'exécutant respectivement sur s_1 et sur s_2 de manière concurrente aux transactions T_1 et T_2 :

$$\begin{array}{rcl} T_3 & = & \{w_3[b], w_3[a]\} \\ T_4 & = & \{w_4[d], w_4[c]\} \end{array} \tag{1.10}$$

Supposons que la transaction T_1 effectue sa première lecture ainsi que T_2. Puis, suit l'exécution sur $s_1$de la première écriture de T_3. Cette transaction est alors forcée d'attendre la levée du verrou posé sur a par T_1 pour effectuer sa deuxième écriture. De même T_4 exécute sur s_2 sa première écriture et est là aussi obligée d'attendre la levée d'un verrou posé cette fois sur c par T_2 pour effectuer sa deuxième écriture. Dès lors, T_1 doit elle aussi attendre la levée d'un verrou posé par T_4 sur d et T_2 doit attendre la levée d'un verrou posé par T_3 sur b. On aboutit ainsi à un verrou mortel, chaque transaction ne pouvant exécuter sa deuxième opération.

Nous avons vu en 1.3.1.3, dans le cas d'un SGBD distribué, deux solutions permettant de résoudre ce problème. La première solution est d'associer à toute transaction un délai de garde au delà duquel la transaction est annulée. Cette solution peut évidemment être appliquée aux SGMB, mais, encore une fois, la détermination du temps à allouer à chaque transaction est très difficile. En effet, un délai trop court peut impliquer l'annulation de beaucoup de transactions simplement non terminées mais non impliquées dans un verrou mortel, entraînant dès lors une perte des ressources. D'un autre côté, un délai trop long résulte en un temps de détection des verrous mortels élevé et contribue à la baisse du nombre de transactions concurrentes, affectant la performance du système.

La deuxième solution décrite dans le cadre des SGBD distribués, repose sur l'exploitation du Wait-For-Graph. Elle ne peut cependant pas être mise en œuvre au sein des SGMB. En effet, la construction du Wait-For-Graph implique la connaissance de toutes les transactions du système. Or, dans un SGMB, les transactions locales sont inconnues du GTM. Une adaptation de la technique Wait-For-Graph pour les SGMB a été proposée dans [Breitbart et al.91]. Il s'agit de construire un graphe dont les nœuds sont des transactions globales. Ce graphe, nommé Potential-Conflict-Graph, contient une arrête entre deux transactions T_i et T_j si et seulement si T_i détient au moins un verrou sur une donnée d'une source de données et T_j attend la levée d'un verrou posé sur une donnée de cette même source de données. Il est facile de voir que s'il y a un circuit dans le Wait-For-Graph, il y a un circuit dans le Potential-Conflict-Graph. L'inverse n'étant pas vrai, le PCG permet de détecter les verrous mortels mais aussi des verrous mortels « virtuels » qui n'entraînent pas le blocage du système. Cette solution n'est donc que partiellement satisfaisante, surtout s'il y a beaucoup de données accédées et distribuées sur peu de sites. Dans ce cas, la proportion des transactions abandonnées à la suite de la détection d'un circuit dans le PCG, par rapport aux transactions émises, peut être élevée. Ainsi, les performances du système risquent de baisser de façon importante et de

devenir équivalentes aux performances du système lors de l'utilisation de la solution « délai de garde » qui est beaucoup plus simple à mettre en œuvre.

Pour terminer, une étude statistique (voir [Baldoni et al.95]) a été menée sur le PCG. Les résultats de cette étude montrent que la majeure partie des verrous mortels sont détectés par un circuit de longueur 2 dans le PCG. Une nouvelle solution peut être déduite de ces résultats : détecter uniquement les circuits de longueur 2 au sein du PCG et poser un délai de garde. Cette approche, dite de détection hybride des verrous mortels, a l'avantage de ne pas obliger la construction du PCG puisqu'il suffit de tester les transactions deux par deux. De plus, la détermination du délai de garde est moins critique que précédemment puisqu'une grande partie des verrous mortels sont détectés à l'aide du PCG.

1.4 Les données du SGMB

1. Entité-Entité
   (a) Conflit un-un
      i. Nom de l'attribut
         A. Des noms différents pour des attributs équivalents
         B. Le même nom pour des attributs différents
      ii. Contraintes
         A. Contraintes d'intégrités
         B. Type de données
         C. Composition
      iii. Valeurs par défauts
      iv. Inclusion d'attributs
      v. Méthodes
   (b) Conflit un à plusieurs

FIG. 1.6 – Classification des conflits d'intégration (1)

Dans les parties précédentes, nous avons présenté l'architecture générale d'un SGMB, c'est-à-dire les différentes couches logicielles ou logiques le constituant. Puis, nous avons décrit les différents problèmes devant être résolus afin de donner aux SGMB de bonnes propriétés transactionnelles. Nous terminons ce chapitre par l'étude du problème le plus important du SGMB, l'intégration des données, soit, comment donner à un utilisateur ou à une application travaillant sur un SGMB, l'illusion d'une seule et même base de données homogène, les données sous-jacentes étant hétérogènes.

Pour résoudre ce problème, il est nécessaire, en sus de la phase de traduction permettant d'homogénéiser les métamodèles de données, de résoudre les différents conflits inhérents à

l'hétérogénéité des sources de données locales dans une phase d'intégration, que ce soit dans le but de lire les données et de les présenter sous une forme homogène ou de mettre à jour les données de l'utilisateur ou de l'application et donc de répercuter les modifications au niveau des sources de données. Si l'on restreint les sources de données locales aux SGBD relationnel ou à objets, il est possible de définir une classification de ces conflits dits structurels (voir [Kim95a]).

1. Conflit Entité-Entité
   (a) Conflit un-un
      i. Nom de l'entité
         A. Des noms différents pour les mêmes entités
         B. Le même nom pour différentes entités
      ii. Structure de l'entité
         A. Attributs manquants
         B. Attributs manquants mais implicites
      iii. Contraintes
      iv. Inclusion
   (b) Conflit un-à-plusieurs
2. Conflits Entité-Attribut
3. Différentes représentations pour des données équivalentes
   (a) Des expressions différentes dénotent la même information.
   (b) Des unités différentes
4. Des niveaux de précision différents.

FIG. 1.7 – Classification des conflits d'intégration (2)

Cette classification (voir les figures 1.6 et 1.7), bien que limitée au cas relationnel et objet, permet de définir deux classes de conflits, indépendantes des métamodèles utilisés dans le SGMB. La première classe comprend les conflits dus à une représentation différente de la même information. La deuxième classe contient les conflits dus à une même représentation d'information distinctes. La résolution de ces deux types de conflits garantit la construction d'un modèle global correct et cohérent. Elle peut être effectuée soit par l'utilisateur lors de requêtes ou lors de la description de correspondances, soit guidée par un module du SGMB qui, ayant connaissance des sources de données locales, propose à l'utilisateur un ou plusieurs modèles globaux (voir [Bright et al.94], [Dixon et al.94], [Dubois97], [Navathe96], ou [Thieme94]).

Dans ce mémoire, nous restreindrons notre étude au cas où l'utilisateur résoud les conflits par lui-même. En effet, on peut considérer que proposer automatiquement ou semi-automatiquement à l'utilisateur un modèle global est une étape supplémentaire, une option qui n'est pas nécessaire à la réalisation d'un SGMB. Nous nous contenterons donc d'étudier d'une part

les différentes possibilités d'intégration de modèles distincts et, d'autre part, les langages de requêtes multibases et leur gestion au sein d'un SGMB.

1.4.1 A propos du métamodèle de données globales

Le choix du métamodèle de données global est un choix critique car les capacités d'intégration de données, d'homogénéisation de métamodèles, et de gestion des requêtes qu'il est possible de soumettre au SGMB ainsi que l'efficacité du système vont en dépendre (voir par exemple [Saltor et al.91]). Nous avons vu en introduction que les données disponibles actuellement à travers le monde sont soit semi-structurées ou non structurées, soit structurées suivant des métamodèles de données très hétérogènes. Parmi les métamodèles utilisés pour modéliser des représentations de ces données, on peut citer les métamodèles relationnel, hiérarchique, réseau ou à objets, et les métamodèles SGML, XML, HTML. Le métamodèle global d'un SGMB doit, si possible, permettre de prendre en compte l'ensemble des capacités d'expression de chacun des métamodèles associés aux sources de données qu'il souhaite intégrer ou même éventuellement qu'il sera souhaitable d'intégrer ultérieurement. Ainsi, le métamodèle à choisir doit être le plus ouvert possible et avoir un important pouvoir d'expression. Une première question à se poser est doit-on utiliser comme métamodèle global l'un des métamodèles associés aux sources de données ou doit-on utiliser un métamodèle conçu uniquement comme métamodèle pivot d'intégration de sources de données hétérogènes. Le premier cas de figure a quelques avantages. D'une part, il permet de faire l'économie de certaines transformations entre le SGMB et les sources de données locales associées au métamodèle global. D'autre part, les utilisateurs du SGMB n'ont pas à faire l'apprentissage d'un nouveau métamodèle et/ou d'un nouveau langage de gestion de données puisqu'ils peuvent utiliser des outils connus et associés au métamodèle global. Les adeptes de la seconde solution répondront à ces avantages que l'utilisation d'un nouveau métamodèle global peut faciliter le processus d'intégration et permettre au SGMB d'être plus efficace même s'il est nécessaire de ne considérer ce nouveau métamodèle que comme un métamodèle pivot et donc de devoir retransformer dans tous les cas les données du SGMB dans le métamodèle de l'utilisateur. Considèrons la figure 1.3 représentant l'architecture en 5 niveaux du SGMB, deux transformations de métamodèles, que nous appellerons Π_1 et Π_2 peuvent être mises en évidence, la première entre le niveau interne et le niveau conceptuel et la seconde entre le niveau conceptuel et le niveau externe. Comme nous venons de le souligner, dans le cas d'un métamodèle global pivot, Π_1 et Π_2 doivent être réalisées quelles que soient les sources de données locales et quels que soient les utilisateurs. Dans le premier cas, Π_1 n'est réalisée que pour les sources de données locales qui ne sont pas structurées suivant le même métamodèle que celui du SGMB et Π_2 n'est réalisée que pour les utilisateurs ne souhaitant pas utiliser ce métamodèle. Pour réduire les coûts de transformations de métamodèles au sein du SGMB, la première approche consiste donc à rechercher un métamodèle qui soit à la fois très utilisé au sein des sources de données locales, ouvert et utilisable tel quel par les utilisateurs du SGMB. La deuxième approche consiste plutôt à rechercher un nouveau métamodèle adapté à l'intégration dont les coûts de transformation de et vers les métamodèles existants soient très faibles.

Actuellement, les travaux suivant la première approche convergent vers l'utilisation d'un métamodèle à objets (voir [Ahmed et al.91], [Lebastard93], [Klas et al.96a], [Roantree et al.99] ou [Springsteel93]) ou relationnel étendu (voir [Agarval et al.92]) alors que ceux concernant la seconde approche proposent l'utilisation d'un métamodèle semi-structuré proche de la théo-

rie des graphes comme XML (voir par exemple [Abiteboul et al.97] et [Simeon99]), ou d'un métamodèle fonctionnel (voir [Machuca et al.94]).

Dans la suite de ce chapitre, nous considérons que la phase de transformation de métamodèle est réalisée. Nous étudions l'intégration de données ainsi que l'exécution de requêtes au sein d'un SGMB dans le cadre d'un métamodèle homogène. Plus particulièrement, nous supposerons que le métamodèle utilisé est un métamodèle à objets. Nous adoptons ainsi la première approche, celle-ci nous paraissant plus proche de nos besoins, tels que décrits en introduction, que la seconde. Nous baserons notre étude sur le modèle à objets suivant, bien adapté au monde des bases de données. Ce métamodèle est une simplification de celui présenté dans [Souza dos Santos95], correspondant au métamodèle du SGBD O2.

1.4.1.1 Un métamodèle à objets

Avant d'être mis en œuvre au sein des systèmes de gestion de bases de données, l'objet est apparu dans le monde de la programmation. Les objets sont alors des entités regroupant les données et les comportements pour les manipuler, et une application est un ensemble d'objets autonomes, capables de communiquer entre eux par ce qu'on appelle des envois de messages. En d'autres termes, chaque objet contient des données qui représentent son état. Cet état peut-être modifié à l'aide des comportements qui lui sont associés. L'état d'une application est l'union de l'état des objets qui la composent.

Pour terminer cette introduction sur l'objet, nous présentons ci-dessous les différents concepts qui lui sont généralement associés.

La classe. Une classe[8] est la description de la structure et du comportement d'un ensemble d'objets. Au sein d'une classe sont décrits d'une part les champs ou attributs des objets, ce qui représente leur structure, et d'autre part, les méthodes. Celles-ci correspondent aux messages pouvant être envoyés aux objets de la classe. Elles forment le comportement de ces objets.

L'instanciation. Comme nous venons de le voir, les classes décrivent les objets. Ces derniers sont donc construits à partir des classes. Cette opération de construction se nomme l'instanciation. Les objets peuvent être d'ailleurs appelés instances de la classe qui à servit à les définir. Enfin, l'instanciation est souvent associée à l'initialisation des champs de l'objet.

L'envoi de message. Un envoi de message est une demande d'exécution d'un certain comportement sur un objet.

L'encapsulation. Les données stockées au sein d'un objet sont cachées pour l'extérieur de l'objet. Pour modifier ou consulter l'état d'un objet, un autre objet ne peut agir que par envoi de message.

[8]Certains préfèrent faire jouer ce rôle au concept de type. La classe représente alors plutôt l'extension du type soit l'ensemble de ses instances.

L'abstraction de données. Il n'est pas nécessaire de connaître la représentation interne d'un objet pour lui adresser un message. Seul l'objet destinataire d'un message peut décider de la manière de traiter celui-ci.

L'héritage. A l'aide de ce concept, chaque classe peut être utilisée pour définir, de manière incrémentale d'autres classes plus spécifiques. Ces nouvelles classes sont appelées sous-classes ou classes filles. On peut aussi voir le concept d'héritage comme la capacité d'une classe mère à factoriser la structure et le comportement de ses sous-classes. Notons que l'héritage peut-être simple, le graphe est dans ce cas une forêt ou un arbre, ou multiple lorsque les classes peuvent hériter de plusieurs super-classes.

Dans la suite de cette partie, nous présentons un modèle formalisant les différents concepts présentés ici basé sur les notations définies dans [Souza dos Santos95].

1.4.1.2 Ensembles

Soit $\mathcal{D}$ l'ensemble infini supposé dénombrable des valeurs atomiques, c'est-à-dire l'union des domaines de type `entier`, `réel`, `booléen`, `chaînes de caractères` et de la valeur `null` représentant l'absence de valeur.

Soit $\mathcal{A}$ l'ensemble infini dénombrable des noms de champs (disjoint de $\mathcal{D}$).

Soit $\mathcal{C}$ l'ensemble infini dénombrable des noms de classes (disjoint des ensembles précédents).

Soit $\mathcal{M}$ l'ensemble infini dénombrable des noms de méthodes (disjoint des ensembles précédents).

Soit $\mathcal{O}$ l'ensemble infini dénombrable des identifiants d'objets (disjoint des ensembles précédents).

Soit O inclus dans $\mathcal{O}$, l'ensemble des valeurs $V(O)$ que l'on peut construire à partir de O est défini par la récurrence suivante (note : $\langle v_1, ..., v_n \rangle$ représente un n-uplet et $\{v_1, ..., v_n\}$ représente un ensemble) :

$$\begin{array}{l} nil \in V(O) \\ \mathcal{D} \subset V(O) \\ O \subset V(O) \\ \forall v_1, ..., v_n \in V(O), \forall a_1, ..., a_n \in \mathcal{A} : \langle a_1 : v_1, ..., a_n : v_n \rangle \in V(O) \\ \forall v_1, ..., v_n \in V(O) : \{v_1, ..., v_n\} \in V(O) \end{array} \tag{1.11}$$

Soit C inclus dans $\mathcal{C}$, l'ensemble des types $T(C)$ que l'on peut construire à partir de C est défini par la récurrence suivante :

$$\begin{array}{l} any \in T(C) \\ \{integer, float, boolean, string, char\} \subset T(C) \\ C \subset T(C) \\ \forall t_1, ..., t_n \in T(C), \forall a_1, ..., a_n \in \mathcal{A} : \langle a_1 : t_1, ..., a_n : t_n \rangle \in T(C) \\ \forall t \in T(C) : \{t\} \in T(C) \end{array} \tag{1.12}$$

1.4.1.3 Hiérarchie de classes, typage et domaine

Une hiérarchie de classe est un triplet $(C, \prec, \sigma)$, où C est un sous-ensemble de $\mathcal{C}$, $\prec$ est une relation d'ordre partielle (réflexive, anti-symétrique et transitive) définie sur les classes de C et σ est une fonction associant à chaque classe $c \in C$ son type élément de $T(C)$.

On supposera que σ définit une forêt soit que l'on se place dans le cas d'un héritage simple où une classe ne peut hériter au plus que d'une seule autre classe.

Le concept de hiérarchie de classes est fortement lié à la notion de spécialisation. En d'autres termes, si une classe c_1 hérite d'une classe c_2, cela implique implicitement que c_1 est plus précise que c_2. Par exemple si c_2 est une représentation des bateaux, c_1 peut être une représentation des bateaux à moteurs. Cela se traduit généralement par une relation entre le type des classes liées par héritage qui peut être modélisée de la manière suivante.

Nous appellerons relation de sous-typage $\leq$ la plus petite relation définie sur $T(C)$ telle que :

$$\begin{array}{l} \forall c, c' \in C : c \prec c' \Rightarrow \sigma(c) \leq \sigma(c') \\ \forall i \in \{1, ...n\}, \forall t_i, t'_i \in T(C), \forall a_i \in \mathcal{A}, : t_i \leq t'_i \Rightarrow \langle a_1 : t_1, ..., a_n : t_n \rangle \leq \langle a_1 : t'_1, ..., a_n : t'_n \rangle \\ \forall t, t' \in T(C) : t \leq t' \Rightarrow \{t\} \leq \{t'\} \\ \forall t \in T(C) : t \leq any \end{array} \tag{1.13}$$

1.4.1.4 Peuplement, Extension et Objets

Nous appellerons peuplement d'une classe c l'ensemble des objets créés spécifiquement dans cette classe soit l'ensemble de ses instances propres. Il ne faut pas confondre la notion de peuplement avec celle d'extension d'une classe qui représente l'ensemble des objets créés dans cette classe ou dans ses sous-classes. De manière plus formelle, définissons une fonction de peuplement π associant à chaque classe l'ensemble de ses instances propres de la façon suivante (note : $P^{fin}(O)$ représente l'ensemble des parties finies de l'ensemble O).

$$\begin{array}{c} \pi : C \rightarrow P^{fin}(O) \\ \forall c, c' \in C, \forall o \in \pi(c) : o \in \pi(c') \Rightarrow c = c' \end{array} \tag{1.14}$$

Cette définition précise que les peuplements associés à deux classes distinctes sont distincts.

A partir de cette définition, il est possible de définir une fonction d'extension $\widetilde{\pi}$ associant à chaque classe son extension.

$$\widetilde{\pi} : \left| \begin{array}{c} C \rightarrow P^{fin}(O) \\ c \mapsto \bigcup \{\pi(c') \mid c' \preceq c\} \end{array} \right. \tag{1.15}$$

Il est maintenant possible de définir le domaine d'un type comme l'ensemble des valeurs pouvant être prise par un objet ou une variable de ce type. Nous appellerons $dom(t)$, le domaine du type t tel que :

- Les types atomiques ont leur domaine usuel.

- $dom(any) = V(O)$
- $\forall c \in C, dom(c) = nil \cup \widetilde{\pi}(c)$
- pour tout entier n, $\forall t_1, ..., t_n \in T(C), \forall a_1, ..., a_n \in \mathcal{A}$, $dom\left(\langle a_1{:}t_1, ..., a_n : t_n\rangle\right) = \{\langle a_1 : v_1, ..., a_n : v_n\rangle \mid v_i \in dom(t_i)\}$

Enfin, définissons la notion d'objet. Un objet est composé d'un identifiant et d'un état correspondant à une valeur prise dans le domaine du type de sa classe. Un objet, instance d'une classe c est donc une paire (o, v) telle que o soit élément de $\pi(c)$ et telle que v soit élément de $dom(\sigma(c))$. Notons que l'on peut changer la valeur d'un objet mais que son identifiant doit rester constant. Deux objets différents peuvent donc avoir la même valeur, mais ont des identificateurs distincts. Nous appellerons v la fonction associant, à un instant donné, à tout identifiant o élément de O une valeur de $dom(\sigma(c))$, c étant la classe de o.

1.4.1.5 Méthodes

Nous appellerons signature d'une méthode de nom m une expression de la forme :

$$m : c \times t_1 \times ... \times t_n \rightarrow t \tag{1.16}$$

Dans cette expression, c est la classe dans laquelle m est définie, t est le type du résultat de la méthode et $t_1, ..., t_n$ sont les types des arguments de la méthode. Notons que l'implémentation de la méthode doit être spécifiée dans un langage de programmation donné. Dans ce chapitre, nous faisons l'abstraction d'un tel langage et nous nous contentons de définir la signature des méthodes.

Une méthode m peut être appliquée à tous les objets instances de la classe dans laquelle elle est définie ainsi qu'à tous les objets des classes qui héritent m de celle-ci. On dira qu'une méthode m est héritée d'une classe c pour c' (et donc que l'on peut appliquer la méthode sur toutes les instances de c') si :

- m n'est pas définie dans c'
- $c' \prec c$ et m est définie dans c et il n'existe pas c'' telle que $c' \prec c'' \prec c$ et telle que m est définie dans c''.

La deuxième condition spécifie qu'une méthode est héritée de la plus petite surclasse où elle est définie.

A un même nom de méthode peuvent être associées plusieurs signatures dans différentes classes, celles-ci pouvant être éventuellement liées par héritage comme nous venons de le voir. Toute signature doit en conséquence vérifier la propriété suivante appelée règle de covariance. Cette règle permet de garantir que toute méthode redéfinie dans une sous-classe d'une classe c soit compatible avec la méthode portant le même nom et définie dans c.

$$\forall m \in \mathcal{M}, \forall c, c' \in C$$
$$(c \prec c') \wedge (m : c \times t_1 \times ... \times t_n \rightarrow t) \wedge (m : c' \times t'_1 \times ... \times t'_n \rightarrow t') \Rightarrow t \leq t' \wedge t_i \leq t'_i, (i = 1, ...n) \tag{1.17}$$

1.4.1.6 Modèle et Base

Le modèle d'une base de données à objets, soit la structure de cette base, est défini par un ensemble de classes associées à leur type et à leurs méthodes, et reliées par une relation d'héritage. Un modèle peut donc être représenté par un 4-uplet $(C, \prec, \sigma, M)$ où M est un ensemble de signatures de méthodes définies sur $(C, \prec, \sigma)$.

Une instance B d'un modèle S, où une base B de S, est définie par un couple (π, υ) où π est une fonction de peuplement pour S et υ une fonction associant à chaque identifiant d'objet o une valeur prise dans $dom(\sigma(c))$, c étant la classe de o.

1.4.2 De l'intégration des données

Le but principal des SGMB est de permettre un accès homogène à de multiples sources de données hétérogènes. Les SGMB sont donc avant tout des outils d'intégration de données. Dans cette partie, nous passons en revue les différentes techniques proposées dans la littérature pour intégrer des données. Nous supposerons que toutes les données sont structurées suivant un métamodèle homogène et que seule subsiste l'hétérogénéité des modèles de données telle que spécifiée dans les figures 1.6 et 1.7. En règle générale, l'accès aux données stockées dans un SGBD consiste à envoyer une requête au système. La requête est formulée en fonction des structures et donc du modèle des données. Comme nous l'avons précédemment mentionné, elle est exprimée dans un langage particulier, associé au métamodèle suivant lequel sont structurées les données, qui permet le plus souvent de décrire précisément l'ensemble des données que l'on souhaite obtenir en fonction de restrictions qu'elles doivent vérifier. Par exemple, considèrons une base de données contenant une représentation de l'ensemble des employés d'une société. Supposons que ces employés soient représentés sous la forme d'une classe appelée `Employe` dont les champs sont le nom, le prénom, le numéro de sécurité sociale et le salaire de l'employé. Il est alors possible de récupérer tous les noms des employés dont le salaire est supérieure à 10000 francs en envoyant au système la requête suivante exprimée en OQL[Cattel et al.97], langage de requêtes utilisé pour accéder aux données de certains SGBD à objets :

```
Select e.nom From Employe e Where e.salaire>10000 ;
```

Il est souhaitable de pouvoir effectuer le même genre de requêtes sur les données d'un SGMB. Ceci implique d'une part de connaître les structures des données virtuellement stockées au sein du SGMB et d'autre part de pouvoir décomposer ces requêtes en autant de partie exécutables par chaque source de données locales et de reconstruire ensuite les résultats pour les présenter à l'utilisateur de manière cohérente et homogène.

Par souci de simplification, certains systèmes tels que celui présenté en [Chen et al.96], ne permettent de décrire le modèle global du SGMB, et donc les structures virtuelles des données du SGMB, que par le biais du langage de requête. Notons que dans ce cas, il n'y a pas vraiment de modèle global puisque l'utilisateur décrit pour chaque requête la forme voulue des données. Le système de gestion des données d'un tel SGMB est ainsi restreint à un interprète de requêtes très complexe. Le principal défaut de cette approche est que chaque utilisateur doit connaître précisément le modèle des sources de données locales afin d'intégrer les données.

Une autre solution, plus proche de l'esprit des SGMB, est d'imposer à un administrateur ou mêmes aux utilisateurs, ou à certains d'entre eux, de définir un ou plusieurs modèles globaux

pour le SGMB. Les requêtes portant sur les données du SGMB sont alors exprimées en fonction d'un modèle du SGMB, le système se chargeant de transformer ces requêtes en requêtes exprimées sur chacun des modèles locaux et de reconstruire les résultats en fonction du modèle global. Dans ce cadre, la description du ou des modèles du SGMB est donc une phase très importante pour la bonne gestion des données intégrées. Pour réaliser cette description, deux approches s'opposent. La première décrite en 1.4.2.1 consiste à décrire les structures du modèle global à réaliser ainsi que les correspondances des différents éléments de ces structures avec ceux des structures des modèles locaux à prendre en compte à l'aide par exemple de requêtes. Nous appelons cette approche, approche déclarative. La deuxième, décrite en 1.4.2.2, consiste à construire le modèle global à partir de transformations élémentaires, bien évidemment virtuelles, des modèles locaux. Nous appelons cette approche, approche transformationnelle.

1.4.2.1 Approche déclarative

L'un des travaux les plus aboutis dans le domaine de l'approche déclarative adaptée au méta-modèle à objets a été réalisée par [Souza dos Santos95]. Ces recherches ont consisté à définir un système de vues appelé O2Views pour le SGBD à objets O2. Une vue dans O2Views est définie comme un modèle virtuel dérivé d'un autre modèle réel ou virtuel. Dans ce système est aussi défini le concept de base virtuelle correspondant à une instance d'un modèle virtuel.

Comme l'on suit une approche objet, toute vue, modèle virtuel, est composée de classes dites virtuelles et/ou de classes dites imaginaires[9], construites de manière descriptive en spécifiant leurs champs ainsi que des requêtes les liant avec des classes du modèle sur lequel est basée la vue, dites classes de base. Les instances des classes virtuelles sont appelées objets virtuels. Chaque objet virtuel ne correspond qu'à une instance d'une classe de base. Les instances des classes imaginaires sont appelées objets imaginaires. Chaque objet imaginaire peut être basé sur plusieurs instances de classes de base éventuellement distinctes.

De façon plus formelle, étant donné un modèle $(C, \prec, \sigma, M)$, une vue est un 5-uplet $(C_{iv}, \prec, \sigma' \cup \sigma_{core}, M')$ dans lequel :

- C_{iv} est inclus dans C et correspond à l'ensemble des noms de classes virtuelles et imaginaires.
- σ' est une fonction de C_{iv} dans $T(C_{iv})$ associant à chaque classe virtuelle ou imaginaire son type.
- σ_{core} est une fonction de C_i (ensemble des classes imaginaires et sous-ensemble de C_{iv}) dans $T(C_i)$ représentant les correspondances entre une classe imaginaire et ses classes de base[10].
- M' est l'ensemble des méthodes définies pour les classes virtuelles ou imaginaires à partir des méthodes de leurs classes de base.

Par analogie avec le concept de base défini en 1.4.1.6, une base virtuelle est un couple (π', υ') dans lequel :

- π' associe à chaque classe virtuelle ou imaginaire ses instances propres telle que pour toute classe $c \in C_{iv}$, $\pi'(c)$ est le résultat d'un calcul[11] portant sur les instances des classes de base. Nous modéliserons ce calcul par une fonction nommée f_i.

[9] Notons que les classes virtuelles définies ici ne correspondent pas aux classes virtuelles de C++, qui ne sont que des classes sans instances.

[10] Cette fonction n'est pas nécessaire pour les classes virtuelles puisque leurs instances ne peuvent être restructurées par rapport aux instances de leurs classes de base.

- v' associe à chaque objet virtuel, instance d'une classe c, une valeur élément de $dom(\sigma' \cup \sigma_{core}(c))$, résultat d'un calcul fonction des instances associées à l'objet virtuel et de la requête décrivant c. Nous modéliserons ce calcul par une fontion nommée f_o.

Notons que le concept de vue est bien adapté à l'intégration de sources de données par une approche déclarative en supposant que les vues correspondant à des modèles globaux sont basées sur un modèle formé de l'union des modèles locaux (voir [Duwairi et al.96] ou [Mak96]).

1.4.2.2 Approche transformationnelle

L'un des travaux fondateurs dans le domaine de l'approche transformationnelle a été réalisée par Motro (voir [Motro87]). Le métamodèle utilisé dans ces travaux est un métamodèle relationnel étendu. Pour décrire les différentes transformations réalisables sur les structures des sources de données locales, nous utiliserons le formalisme présenté en 1.4.1.

Le but des travaux de Motro est de définir divers opérateurs minimaux de transformation permettant de résoudre incrémentalement les conflits décrits au début de la partie 1.4 et ainsi d'accéder de manière homogène, au moins en lecture, aux données des sources de données locales. Pour construire le modèle global, on applique les opérateurs choisis sur une union virtuelle des modèles locaux. Les différents opérateurs sont résumés ci-dessous.

Meet . Cet opérateur permet de combiner les propriétés communes de deux classes en créant l'équivalent d'une superclasse. Plus formellement, soit c et $c' \in C$, $meet(c, c') = c'' \in C$ telle que $\sigma(c'') = \sigma(c) \cap \sigma(c')$ et telle que $\widetilde{\pi}(c'') = \widetilde{\pi}(c) \cup \widetilde{\pi}(c')$.

Join *Join* est l'opérateur dual de *meet*, c'est-à-dire qu'il crée à partir de deux classes c et c' une nouvelle classe $c'' \in C$ telle que $\sigma(c'') = \sigma(c) \cup \sigma(c')$ et telle que $\widetilde{\pi}(c'') = \widetilde{\pi}(c) \cap \widetilde{\pi}(c')$. Cet opérateur est identique à l'opérateur de jointure relationnelle et nécessite que les deux classes c et c' contiennent une information commune.

Fold *Fold*, similaire à *meet* et *join*, permet de créer la généralisation deux classes, sachant que le type de l'une d'entre-elle est un sous-type du type de l'autre. En d'autres termes, soit c et c' telles que $\sigma(c') \leq \sigma(c)$, $fold(c_{av}, c') = c_{ap}$, c_{av} et c_{ap} désignent la classe c respectivement avant et après l'opération. Le résultat « visible » de l'opération est $\widetilde{\pi}(c_{ap}) = \widetilde{\pi}(c_{av}) \cup \widetilde{\pi}(c')$.

Rename Ce opérateur crée à partir d'une classe c une classe c' dont seul le nom diffère de celui de c. Plus formellement $rename(c) = c'$ telle que $\sigma(c') = \sigma(c)$ et telle que $\widetilde{\pi}(c') = \widetilde{\pi}(c)$. On suppose que l'opération a pour effet de bord de supprimer la classe c.

Aggregate Cet opérateur crée à partir d'une classe c et d'un type $t_s \leq \sigma(c)$ une nouvelle classe c' de type t_s. Il modifie le type de c en y remplacant t_s par une référence à c'. Si l'on considère que les types de c et de t_s ne peuvent être que des n-uplets, c'est-à-dire $\sigma(c) = \langle a_1 : t_1, ..., a_n : t_n \rangle$ et $t_s = \langle a_m : t_m, ..., a_n : t_n \rangle$ avec $1 < m < n$, on a $aggregate(c, t_s) = c'$ telle que $\sigma(c') = t_s$ et $s(c) = \langle a_1 : t_1, ..., a_{m-1} : t_{m-1}, a''_m : c' \rangle$.

[11] Ce calcul peut être l'identité dans le cas de classes virtuelles.

Telescope Cet opérateur est le dual de l'opérateur précédent. Soit deux classes c et c' telles que $\sigma(c) = \langle a_1 : t_1, ..., a_{m-1} : t_{m-1}, a_m : c' \rangle$ et $\sigma(c') = \langle a'_1 : t'_1, ..., a'_n : t'_n \rangle$, $fold(c_{av}, c') = c_{ap}$ telle que $\sigma(c_{ap}) = \langle a_1 : t_1, ..., a_{m-1} : t_{m-1}, a'_1 : t'_1, ..., a'_n : t'_n \rangle$, c_{av} et c_{ap} désignent la classe c respectivement avant et après l'opération.

Add Cet opérateur ajoute au modèle global une nouvelle classe c' ayant une unique instance et rajoute à une classe c une référence vers c'. Ajouter une information non existante dans le modèle global ne peut généralement être considéré que comme l'ajout d'une nouvelle source de données locale et non comme une opération de restructuration. Néanmoins, dans certains cas, des classes peuvent avoir des attributs implicites qu'il peut-être intéressant d'expliciter pour faciliter l'intégration. Par exemple, supposons que nous souhaitions intégrer deux bases de voitures, l'une contenant la représentation sous forme d'une classe des voitures de marque Renault et l'autre la représentation sous forme d'une autre classe des voitures de marque Peugeot. Lors du processus d'intégration regroupant ces deux classes, il paraît intéressant de rajouter un champ marque dans la classe résultante pour différencier les deux types de voiture. Ceci peut être réalisé à l'aide de *add*.

Delete Cet opérateur permet de supprimer les classes du modèle virtuel non nécessaires à la réalisation du modèle global.

1.4.2.3 Mises à jour

Les deux approches décrites précédemment conviennent parfaitement à l'exploitation homogène, en lecture, de données distribuées au sein de sources de données hétérogènes. En effet, toute la démarche de construction du modèle global repose sur la description de fonctions, d'opérateurs ou de requêtes permettant de calculer les objets du SGMB en fonction des objets des sources de données locales. L'un des objectifs des SGMB est de se comporter comme un SGBD normal. En particulier, il est souhaitable de permettre aux utilisateurs du SGMB de mettre à jour les données. Nous avons vu dans la partie 1.3.2 quels étaient les problèmes liés à cette liberté sur le système transactionnel du SGMB. D'autres problèmes surviennent au niveau du gestionnaire de données. En effet, mettre à jour les données virtuelles du SGMB implique, pour respecter les objectifs de ces systèmes, de mettre à jour en temps-réel les données sous-jacentes stockées au sein des sources de données locales. Deux types de problèmes se posent alors :

- Comment transformer les mises à jour des données intégrées exprimées dans le métamodèle global en mises à jour de données locales exprimées dans le métamodèle global.
- Comment transformer les mises à jour de données locales exprimées dans le métamodèle global en mises à jour de données locales exprimées dans le métamodèle local.

Le premier type de problème revient à chercher à inverser les fonctions f_o et f_i si l'on se place dans la première approche ou les opérateurs de transformation de la seconde approche. Dans le second cas, il peut être assez complexe, voire dans certains cas impossible de réaliser l'inversion en fonction des opérateurs choisis. Dans le premier cas, l'inversion de la fonction f_i est simple pour peu que l'on choisisse une technique de calcul inversible ce qui n'a pas d'influence sur la généralité et la puissance du système de vue (voir par exemple [Damodoran-Kamal et al.94]

ou [Bellahsene97]). A l'inverse, il est clair que l'inversion de f_o est en règle générale non triviale, voire impossible. De nombreux travaux ont été réalisés sur ce sujet dans le cadre d'un métamodèle relationnel (voir par exemple [Cosmadadis et al.84]) et permettent d'effectuer de nombreuses mises à jour même complexes. Dans le cadre d'un métamodèle à objets, encore peu de travaux ont été réalisés, mais il en ressort que les mises à jour peuvent être effectuées dans les cas simples, et lorsque l'utilisateur décrit lui-même les fonctions inverses (voir par exemple [Amer-Yahia et al.97] ou [Gentile94]).

Le second type de problème est lui-aussi non trivial. De nombreux travaux ont été réalisés sur ce sujet par exemple en ce qui concerne le passage du métamodèle relationnel à un métamodèle à objets (voir par exemple [Lebastard93] ou [Delobel et al.95]). Dans ce cas aussi, les possibilités de mises à jour dépendent du contexte et des transformations réalisées pour obtenir le modèle global.

1.4.3 A l'exécution des requêtes utilisateurs

En se référant à l'architecture à 5 niveaux (voir la figure 1.3), on voit que l'exécution d'une requête globale, traitée par le SGMB, s'effectue en trois étapes si l'on occulte le niveau externe. Tout d'abord, la requête, exprimée dans le langage de requêtes associé au métamodèle global est décomposée en sous-requêtes, dites globales, telles que les données nécessaires à l'exécution de chaque sous-requête soient stockées dans une seule source de données locale. Ceci nous fait passer au niveau traduit, les sous-requêtes étant toujours exprimées dans le langage associé au métamodèle global. Puis, chaque sous-requête est traduite dans le langage associé au métamodèle local correspondant, ce qui nous place au niveau composant, avant d'être envoyée à la source de donnée locale pour son exécution. Enfin, les résultats de ces sous-requêtes, dites locales, sont recomposées en suivant le chemin inverse. Notons qu'il est assez facile d'étendre cette technique pour prendre en compte le niveau externe du SGMB.

De façon générale, la première étape de cette technique peut elle-même être décomposée en deux étapes. Tout d'abord, la requête globale qui se réfère aux noms et structures du modèle globales est modifiée ou transformée en une requête identique mais se référant, cette fois, aux noms et structures locales correspondantes et définies dans le modèle exporté. Notons qu'une telle requête peut contenir des références à des données stockées dans plusieurs sources de données locales. La deuxième étape est constituée de la décomposition à proprement parlé. Notons que cette étape peut être combinée avec l'exécution partielle de requêtes. Par exemple, supposons que la requête transformée soit une requête imbriquée pouvant être décomposée en une requête interne et une requête externe. Il est possible que le résultat de la première requête soit nécessaire à la décomposition et/ou à l'exécution de la seconde. Il est alors nécessaire de permettre l'exécution de requêtes simultanément à l'algorithme de décomposition.

L'étape de décomposition étant terminée, il reste, comme nous l'avons souligné plus haut, à traduire les sous-requêtes générées dans le langage associé au métamodèle de la source de données locale correspondante. Notons qu'une sous-requête globale peut être traduite en une ou plusieurs sous-requête locale. Il est aussi possible qu'une sous-requête globale ne puisse être traduite dans le langage de requêtes local, car certaines fonctionalités du langage global ne peuvent être exprimées dans le langage global. Dans ce cas, la sous-requête globale est traduite en une sous-requête locale plus large, dont le résultat inclut le résultat recherché.

Cela implique que le gestionnaire de requêtes globale puisse ensuite filtrer le résultat de la sous-requête locale pour finalement donner le bon résultat.

La complexité et la forme précise de l'algorithme de décomposition et d'exécution de requêtes globales dépend en fait de nombreux critères parmis lesquels :

- le langage de requêtes et le métamodèle global ;
- la méthode utilisée pour intégrer les modèles locaux. En règle générale, il sera plus facile de définir un algorithme de décomposition et de recomposition lorsque cette méthode correspond à une approche transformationnelle. (voir par exemple [Kapsammer et al.97], [Meng et al.95] ou [Florescu et al.95]) ;
- si les sources de données locales contiennent des données représentant les mêmes données du monde réel ou non ;
- s'il existe des incohérences entre plusieurs sources de données locales. Par exemple, il se peut que deux sources de données locales modélisent la même entité du monde réel, mais que l'une, voire les deux, disposent d'information erronées différentes. (voir par exemple [Tomasic et al.95] ou [Kapitsaia et al.97]).

Plus généralement, plus les sources de données sont en conflit, par rapport à ce que nous avons décrit au début de la partie 1.4, et plus l'algorithme sera complexe.

Enfin, pour terminer cette partie, notons que l'optimisation de requêtes globales et plus généralement de l'exécution de requêtes au sein d'un SGMB est un problème non-trivial. Celui-ci est fortement lié à celui de l'optimisation au sein de SGBD distribués. Dans ce cas, de nombreuses solutions sont décrites dans la littérature (voir par exemple [Oszu et al.99]). Celles-ci imposent cependant qu'il n'y ait pas d'inconsistance entre les données (c'est-à-dire que les problèmes liés à l'intégration sémantique ne sont pas pris en compte), que les sources de données locales puissent communiquer, ou que chaque source de données locale puisse fournir des statistiques précises sur l'accès aux données. Or, ces contraintes ne peuvent être vérifiées dans le cadre d'un SGMB à cause des autonomies DEC. En effet, dans ce cas, on doit supposer que les sources de données locales ne peuvent communiquer entre elles et donc coordonner leurs optimiseurs locaux, et encore moins communiquer au SGMB des mesures de performance sur l'accès à leur données, le SGMB devant être considéré comme un utilisateur normal. Pour résoudre le problème dans le cadre des SGMB, certains travaux proposent de créer un moniteur chargé de calculer le temps d'exécution de requêtes « échantillons » afin de calibrer l'optimisation de l'exécution des requêtes (voir [Lipton et al.90], [Gardarin95] ou [Zhu et al.96]). Ces techniques doivent ainsi permettre de rendre le SGMB globalement plus efficace au fur et à mesure des exécutions.

Chapitre 2

Des SGMB aux langages réflexifs à objets

2.1 Analyse

En règle générale, les utilisateurs d'informations stockées au sein de SGMB, ou plus généralement au sein de sources de données, n'exploitent les données qu'à travers des applications servant d'intermédiaires avec ces systèmes. En ce sens, nous considérons dans ce chapitre que les programmeurs de ces applications constituent les véritables utilisateurs des SGMB, les autres étant relégués au rang d'utilisateurs indirects.

Cette approche nous permet d'avoir un regard nouveau sur les problèmes soulevés dans le chapitre précédent. Dans cette partie, nous nous intéressons aux conséquences du positionnement de ces problèmes du point de vue du programmeur. En particulier, nous redéfinissons le concept d'hétérogénéité et nous étudions les différents modes d'exploitation de sources de données pouvant être mis en œuvre par les programmeurs. Cette étude nous permettra de déterminer l'approche idéale, à notre sens, de développement d'applications accédant à de multiples sources de données hétérogènes et indépendantes. Nous en déduirons des pistes de modélisation d'un système de développement adapté à cette approche.

2.1.1 A propos de l'hétérogénéité

Dans le chapitre précédent, nous avons montré qu'il était possible de présenter une vue unifiée d'un ensemble de sources de données hétérogènes et indépendantes à l'aide des SGMB. Ces systèmes permettent de résoudre ce que nous appelons l'hétérogénéité de stockage des données. Ce concept regroupe les deux types d'hétérogénéité que nous présentons ci-dessous.

l'hétérogénéité de stockage système : Ce type d'hétérogénéité provient d'une part de l'hétérogénéité des systèmes matériels sur lesquels s'exécutent les sources de données à intégrer, que ce soit au niveau de l'architecture matérielle ou du système d'exploitation. D'autre part, il résulte de l'hétérogénéité des réseaux interconnectant les différents sites impliqués et sur lesquels transitent les données.

l'hétérogénéité de stockage conceptuelle : Ce type d'hétérogénéité a été présenté en détail dans le chapitre précédent. Il provient d'une part de l'hétérogénéité des modèles et des métamodèles suivant lesquels sont structurées les données à intégrer, et d'autre part, de l'hétérogénéité des outils et des techniques à l'aide desquels ces données peuvent être exploitées.

Comme nous l'avons précédemment souligné, les travaux cherchant à résoudre les problèmes liés à l'hétérogénéité de stockage sont issus du besoin de voir un ensemble de sources de données de façon homogène. À l'image de l'architecture en 5 niveaux des SGMB telle que décrite dans le chapitre précédent, cette vision du problème de l'hétérogénéité est une vision ascendante. On part de systèmes hétérogènes et on monte par étape vers une vue unifiée et homogène de ces systèmes. Il est possible de prendre le problème dans l'autre sens. On part alors non plus d'un besoin unique des programmeurs de voir un ensemble de sources de données hétérogènes comme un seul et même SGBD, mais de besoins hétérogènes de gestion d'un ensemble d'informations hétérogènes. Cela sous-entend que l'on ne considère plus un ensemble d'outils et de données persistantes hétérogènes en essayant de l'exploiter d'une manière homogène et identique pour tous les utilisateurs du système. On considère plutôt, les besoins de chaque programmeur, en termes de données ou d'outils, en essayant de les mettre en correspondance avec les outils et les données disponibles éventuellement au sein de sources de données hétérogènes (voir par exemple [Uniface]). Cela conduit à essayer de résoudre un nouveau type d'hétérogénéité que nous appelons l'hétérogénéité d'accès. Ce concept regroupe :

l'hétérogénéité de nature des données : Le développement d'application peut nécessiter la mise en œuvre de trois types de données, les données persistantes stockées au sein de sources de données, les données distribuées réparties sur plusieurs sites et les données volatiles, ni persistantes, ni distribuées. Il est souhaitable de permettre une gestion homogène de ces différentes natures de données. En ce sens, nous élargissons le concept de sources de données aux mémoires volatiles des ordinateurs et aux processus distribués distants.

l'hétérogénéité d'accès conceptuelle : Ce concept est l'équivalent de celui développé pour l'hétérogénéité de stockage mais déplacé au niveau du programmeur. Il est en effet possible que chaque programmeur souhaite utiliser dans le but d'exploiter un même ensemble de données, ses propres structures et/ou formalismes. Cela revient à résoudre l'hétérogénéité des modèles et métamodèles des programmeurs en fonction des besoins de chaque programmeur. De même, il est souhaitable que chaque programmeur puisse utiliser ses propres techniques et outils pour accéder et gérer ces données quels que soient leur nature, leur lien de stockage effectif, etc ...

Il est clair que les SGMB résolvent le problème de l'hétérogénéité de stockage. Cela fait, par ailleurs, partie de leurs objectifs. Cependant, dans le meilleur des cas, ces systèmes n'apportent qu'une solution partielle aux problèmes associés à l'hétérogénéité d'accès. Dans certains cas, il est même possible qu'ils contribuent à l'augmenter puisqu'en règle générale, ils cherchent à donner une vision homogène pour tous leurs utilisateurs d'un ensemble de sources de données hétérogènes et donc obligent ceux-ci à utiliser le modèle, le métamodèle et les outils de gestion du SGMB pour accéder aux données unifiées. Par exemple, supposons qu'un SGMB intègre deux sources de données A et B, sachant que A permet l'accès à ses données à l'aide d'un système transactionnel basé sur le modèle des transactions imbriquées, alors que le système

transationnel de B ne peut mettre en œuvre que des transactions plates. Supposons de plus, que le système transactionnel du SGMB ne soit lui aussi conçu que pour gérer des transactions plates. Tous les programmeurs souhaitant développer des applications, accédant à la fois aux données de A et de B, à l'aide du SGMB doivent donc nécessairement utiliser des transactions plates, même s'il aurait été plus approprié d'utiliser des transactions imbriquées et que cela aurait été le cas s'ils n'avaient eu besoin que des données stockées au sein du système A. Prenons un autre exemple. Il est très probable que le modèle global d'un SGMB fédéré simple soit défini a priori, sans prendre en compte les besoins des utilisateurs du SGMB. Dans ce cas, le modèle global est souvent le résultat d'un compromis entre la taille du modèle, qui influe sur les performances générales du SGMB, et les besoins supposés des utilisateurs. Les applications développées à l'aide d'un tel SGMB peuvent ainsi ne pas pouvoir exploiter totalement la richesse potentielle des informations stockées.

Les deux types d'hétérogénéité définis dans cette partie semblent donc incompatibles puisque résoudre l'un implique dans certains cas augmenter l'autre. Néanmoins, nous montrons plus loin comment résoudre simultanément ces deux types d'hétérogénéité.

2.1.2 Notes sur les données volatiles et distribuées

Nous avons affirmé ci-dessus qu'il était intéressant d'avoir la possibilité de gérer à l'identique données volatiles, persistantes et distribuées. Cela sous-entend en particulier qu'il est intéressant d'appliquer des techniques utilisées généralement lors de la gestion de données persistantes aux donnée volatiles et aux données distribuées. Il nous semble utile de nous arrêter quelques instants sur ce point en présentant quelques exemples vérifiant cette proposition (voir [Gray81] et [Guerraoui96]). Pour ce faire, dans cette partie, nous mettons l'accent sur le concept de transaction en étudiant deux utilisations d'un gestionnaire transactionnel, l'une adaptée aux données volatiles et l'autre aux données distribuées.

2.1.2.1 Contrôle de concurrence en programmation distribuée

Même si les techniques de programmation par objets ont contribué à le simplifier, le développement d'applications distribuées est encore aujourd'hui très complexe. La raison en est simple : les langages de programmation actuels n'ont pas été conçus dans ce but et ne proposent que peu d'outils de gestion de distribution. De plus, même si de nombreux travaux de recherche se sont penchés sur ce thème (voir par exemple [Detlef et al.88] et [Liskov88]), ils se sont concrétisés par de nouveaux langages intrinsèquement distribués mais non adaptés aux réalités du développement.

Enfin, les nouvelles architectures et/ou standards tels que l'Integrated Systems Architecture (ISA) [APM91], l'Open Software Foundation Distributed Computing Environment [OSF91] ou l'Object Management Group Common Object Request Broker (CORBA) [OMG97] favorisent le développement d'applications distribuées mais leur mise en œuvre nécessite toujours un effort important. De nouveaux travaux de recherche s'orientent en conséquence vers la définition de systèmes permettant d'intégrer le concept de distribution aux applications de manière « quasi » transparente, c'est-à-dire[1] (voir [Parrington et al.88] et [Parrington92]) :

[1] Notons que ces objectifs sont aussi ceux de CORBA2

1. Sans connaître l'endroit exact où se trouvent les objets distants manipulés, ceux-ci pouvant éventuellement migrer d'un endroit à un autre.

2. En accédant aux objets de la même manière qu'ils soient locaux ou distants.

3. Sans que d'éventuelles fautes dues au réseau et impliquant des exécution partielles n'aient d'effets sur le système.

4. Sans se soucier de la concurrence intrinsèque à la distribution des objets.

Pour en revenir aux transactions, il apparaît clairement qu'un système transactionnel peut-être utile pour développer de tels travaux, principalement en ce qui concerne les points 3 et 4. En effet, le point 3 sous-entend qu'il est nécessaire de gérer l'atomicité des exécutions distantes puisque les exécutions partielles ne doivent pas avoir d'effets sur le système. Quant au point 4, il implique le respect des propriétés de concurrence et d'isolation telles que définies dans le chapitre précédent. Notons que cette approche peut être étendue aux données volatiles dans le contexte du développement d'applications multithreads. Dans ce cas, il ne s'agit plus de garantir la cohérence des données distribuées sur plusieurs processus mais des données volatiles pouvant être accédées par plusieurs threads.

2.1.2.2 Gestion des erreurs

Le concept d'exception a été introduit dans les langages de programmation pour clarifier la programmation et séparer le code d'exécution normale du code de gestion des erreurs. Or, dans certains cas, cette séparation est impossible. Prenons un exemple. Supposons que nous voulions réaliser un programme combinant deux fichiers textes sources en un seul fichier cible par exemple en copiant alternativement une ligne de chaque fichier source dans le fichier cible. Décrivons un algorithme correspondant à ce programme en pseudo-Java (voir la figure 2.1). Nous supposerons que Fs1 et Fs2 représentent les fichiers sources et Fc le fichier cible.

```
Fs1.open() ;
Fs2.open() ;
Fc.open() ;

Fc.concatene(Fs1,Fs2) ;

Fs1.close() ;
Fs2.close() ;
Fc.close() ;
```

FIG. 2.1 – Gestion des erreurs : Exemple simple

Les trois premières opérations ouvrent les fichiers, la quatrième réalise l'opération souhaitée, et les trois dernières referment les fichiers. Le code décrit ci-dessus ne tient pas compte des erreurs éventuelles générées lors de l'ouverture des fichiers. Pour en tenir compte, en supposant que le langage de programmation utilisé gère les exceptions, nous modifions légèrement le code décrit précédemment (voir la figure 2.2). Dans ce cas, les opérations d'ouverture de fichier et de concaténation correspondant au comportement normal de l'application à réaliser

```
try { // Exécution Normale
        Fs1.open() ;
        Fs2.open() ;
        Fc.open() ;

        Fc.concatene(Fs1,Fs2) ;
}
catch (Error e) { // bloc exécuté s'il y a une erreur
        ...
}
finally { // bloc exécuté dans tous les cas
        if (Fs1.IsOpen())
                Fs1.close() ;
        if (Fs3.IsOpen())
                Fs2.close() ;
        if (Fc.IsOpen())
                Fc.close() ;
}
```

FIG. 2.2 – Gestion des erreurs : exceptions

sont encapsulées dans un bloc `try`. Si une erreur se produit dans ce bloc, le bloc catch, qui correspond au traitement des erreurs, est exécuté . Dans tous les cas, qu'il y ait une erreur ou non, il est nécessaire de fermer les fichiers ouverts. Les opérations nécessaires sont encapsulées dans le bloc finally. De premier abord, cette manière de traiter les erreurs est assez intéressante car elle semble séparer le comportement normal de l'application du comportement en cas d'échec. Néanmoins, dans notre exemple, il est nécessaire dans les deux cas de fermer les fichiers ouverts ce qui implique de tester l'état des fichiers. En effet, si une erreur se produit pendant l'ouverture d'un fichier, il est probable que tous les fichiers ne soient pas ouverts. En conséquence la séparation entre comportement normal et comportement en cas d'échec n'est plus nette, d'autant plus que si l'on considère que les opérations de fermeture de fichiers peuvent échouer, il est alors nécessaire de rajouter des blocs `try`, `catch` et `finally` dans le bloc `finally` et ainsi de suite.

Reprenons le même algorithme dans un cadre transactionnel en supposant que les transactions vérifient la propriété d'atomicité. Pour plus de clarté, nous utiliserons les comportements transactionnels classiques : `begin` pour démarrer une transaction, `commit` pour la valider et `abort` pour l'annuler (voir la figure 2.3).

Cette fois, la séparation entre exécution normale et gestion des erreurs est nette. Notons que l'exécution est tout aussi cohérente que précédemment puisque lorsque la transaction annule, l'application retourne dans l'état précédent le début de la transaction (vérification de la propriété d'atomicité).

```
Transaction trans = new Transaction() ;
trans.begin() ;
try {
        Fs1.open() ;
        Fs2.open() ;
        Fc.open() ;

        Fc.concatene(Fs1,Fs2) ;

        Fs1.close() ;
        Fs2.close() ;
        Fc.close() ;

        trans.commit() ;
}
catch (Error e) {
        ...
        trans.abort()
}
```

FIG. 2.3 – Gestion des erreurs : transactions

2.1.3 Couplages entre langages de programmation et SGBD

Comme nous nous restreignons à l'étude des interactions entre les programmeurs et les SGMB, nous présentons dans cette partie les différentes approches utilisées lors du développement d'applications gérant des données stockées ou persistantes, appelées applications persistantes. En d'autres termes, nous étudions les différents couplages possibles entre les langages de programmation, outils du programmeur, et les systèmes de stockage de données (voir la figure 2.4).

2.1.3.1 Couplage faible

L'approche par couplage faible correspond à une architecture dans laquelle les systèmes de stockage sont extérieurs au langage de programmation. Dans ce contexte, une Interface de Programmation (A.P.I.) transactionnelle plus ou moins complexe et souvent normalisée (voir par exemple OTS [OMG97], JTS [Cheung et al.98], OSI TP [ISO90]) est exportée par chaque source de données vers le langage de programmation. Elle permet de créer, de valider, d'annuler des transactions et parfois même de gérer explicitement la concurrence des données persistantes. De même, une autre A.P.I. permet d'exprimer des requêtes depuis le langage de programmation, de les envoyer vers la source de données et de traiter les résultats. Cette approche permet de disposer de systèmes de stockage de données complètement indépendants du langage de programmation. Un seul gestionnaire de transactions, de requêtes et de données suffit pour gérer des accès multi-applications et concurrents à celles-ci. En contrepartie, la gestion des données persistantes au sein du langage est complexe et n'est pas forcément cohérente et uniforme avec celle des données volatiles. Elle nécessite l'apprentissage du ou des

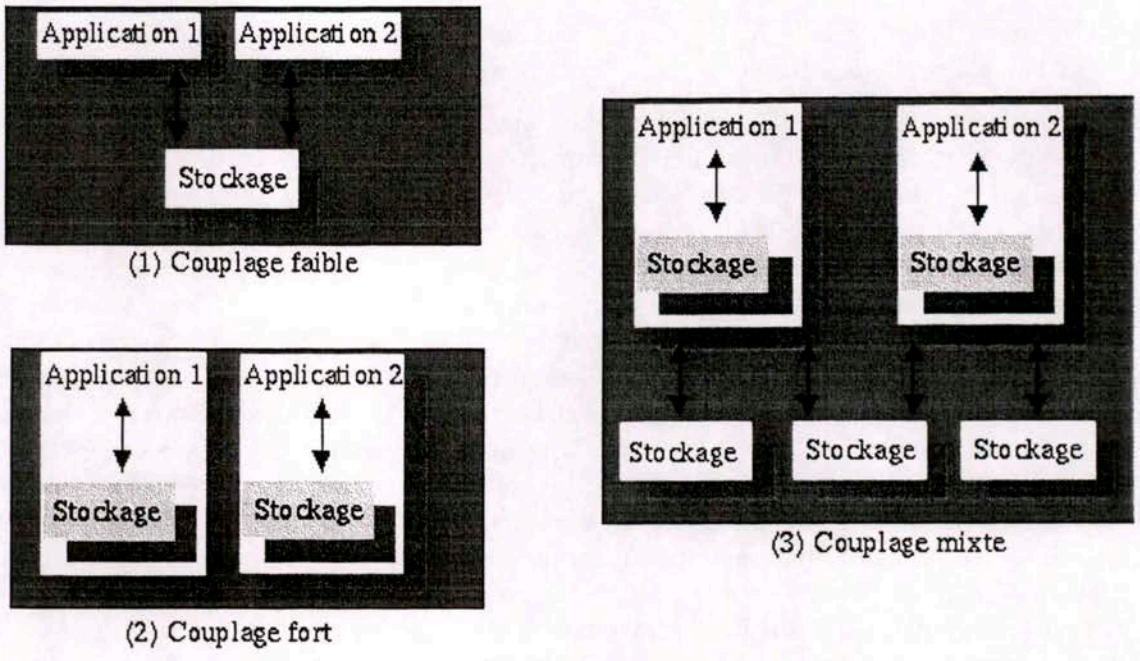


FIG. 2.4 – Types de couplage

langages de requêtes permettant d'accéder aux données persistantes et du formalisme transactionnel associé. En conséquence, le programmeur ne doit pas uniquement se concentrer sur la sémantique de son application mais aussi sur la manière de traiter chaque donnée en fonction de sa nature.

Ce couplage est celui mis en œuvre lors de l'utilisation de SGBD ou de SGMB. Ces deux types de système ont en effet pour but le partage multi-applications des données, virtuelles ou non, qu'ils contiennent. Il est de plus très bien adapté à la résolution de l'hétérogénéité de stockage présentée précédemment car il s'inscrit plutôt dans le cadre d'une approche ascendante où l'on souhaite exploiter des données existantes, ce qui implique là encore, que ces données soient stockées dans un système permettant des accès multi-applications concurrents.

2.1.3.2 Couplage fort

L'approche duale de celle présentée ci-dessus, dite approche par couplage fort, consiste à intégrer le système de stockage au langage de programmation. Dans ce cas, la gestion transactionnelle ou le traitement des requêtes fait partie de la sémantique du langage, par exemple par le biais de mots-clés dédiés. Il est alors possible de gérer les données à l'identique indépendamment de leur nature. De plus, dans certains cas, il peut aussi être possible de spécialiser les comportements associés à la gestion des transactions ou des requêtes pour qu'elle corresponde exactement aux besoins de l'application. Comme ces modifications éventuelles prennent place au sein du langage, elles peuvent n'être valables que pour l'application concernée, ce qui a l'avantage de ne pas avoir de conséquence sur le traitement des données au sein d'autres applications.

Les inconvénients de cette approche sont les avantages de la précédente. Dans ce cas, bien

évidemment, le système de stockage n'est indépendant ni du langage, ni de l'application. La capacité d'un partage « universel » des données n'est en effet pas réalisable dans ce contexte.

Ce type de couplage est utilisé lors du développement d'applications nécessitant la possibilité de rendre persistants les objets qui les constituent non dans le but de partager ces informations avec d'autres applications mais, soit pour conserver des informations entre plusieurs exécutions, soit pour garantir une relative tolérance aux pannes de l'application ou du système d'exploitation. Les systèmes de gestion de bases de données répondent bien évidemment aux besoins de telles applications. Cependant, comme une grande partie de leurs fonctionnalités, telles que la gestion des requêtes, le contrôle de concurrence, la gestion de vues, ou la gestion multi-utilisateurs est, dans ce cadre, sous-exploitée ou même inutilisée, on leur préfère généralement des outils plus simples et necessitant un investissement beaucoup moins important. Nous regroupons ces outils sous le terme d'outils de persistance immédiate car ils cherchent à minimiser le coût de développement de la persistance au sein des applications. Ces outils sont généralement des langages de programmation persistants ou des extensions persistantes de langages de programmation (voir [Atkinson et al.96b]). Ils sont pour la plupart très bien adaptés à la résolution de l'hétérogénéité d'accès lorsqu'ils permettent la spécialisation des comportements transactionnels, de persistance, et/ou de gestion de requêtes. Ils s'inscrivent en effet plutôt dans le cadre d'une approche descendante où l'on souhaite mettre en œuvre la persistance des entités de l'application en prenant en compte de manière optimale les besoins de l'application.

2.1.3.3 Couplage mixte

Pour combiner les avantages et les propriétés des deux approches précédemment présentées, il est possible d'utiliser une approche par couplage mixte. Celle-ci est similaire à l'approche par couplage fort à laquelle il est ajoutée la propriété de partage universel propre à l'approche par couplage faible. Pour ce faire, il est permis au système de stockage intégré au langage de programmation de déléguer, lorsque cela est nécessaire, une partie du traitement transactionnel ou des requêtes à des systèmes de stockage externes. Il est alors possible d'accéder à des données stockées indépendamment du langage, de même que de stocker des données pouvant être exploitées par d'autres systèmes, et ceci en conservant les avantages du couplage fort, c'est-à-dire en respectant la sémantique du langage, en adoptant une approche homogène quelle que soit la nature des données, et en permettant la spécialisation des comportements transactionnels ou de gestion de données en fonction des besoins de l'application.

Ce type de couplage est une solution pour résoudre conjointement l'hétérogénéité de stockage et l'hétérogénéité d'accès. En effet, le programmeur mettant en œuvre cette approche dispose d'un outil capable de prendre en compte de manière optimale à la fois les besoins de son application, les besoins de partage des données et les besoins d'intégration de données existantes.

2.2 Quelques pistes de modélisation

L'objectif de nos travaux est de définir un système de développement, associant un langage de programmation et un système de stockage par couplage mixte, dans le but de résoudre à la fois les problèmes posés par l'hétérogénéité de stockage et l'hétérogénéité d'accès. Ce système

doit permettre de manipuler et de gérer à l'identique et suivant le modèle de l'utilisateur des données volatiles ou persistantes, celles-ci pouvant être stockées dans des sources de données distribuées et hétérogènes. Il doit de plus être complètement paramétrable pour garantir une parfaite adéquation avec les applications développées. Notons que nous restreignons notre champs d'étude aux seules données volatiles ou persistantes ainsi qu'au cas où l'utilisateur ne souhaite pas utiliser un autre métamodèle que celui proposé au sein de notre système. Nous verrons plus loin que grâce aux propriétés d'ouverture et de spécialisation de notre système, ces deux restrictions peuvent être assez facilement levées.

2.2.1 Persistance, transactions, requêtes et vues.

Dans le contexte d'une approche par couplage mixte, l'association d'un langage de programmation et d'un système de stockage de données peut être effectuée par l'ajout au langage des outils, des techniques et des concepts nécessaires à la gestion des données persistantes. Dans un contexte d'intégration de sources de données hétérogènes, il semble intéressant que ceux-ci soient adaptés des SGMB, systèmes de référence de ce domaine. En effet, par ce biais l'hétérogénéité de stockage sera résolue. Par ailleurs, en ouvrant et en rendant spécialisables les différents composants ajoutés de cette manière au langage, nous nous assurons de pouvoir résoudre aussi l'hétérogénéité d'accès aux données. En conséquence, nous souhaitons ajouter au langage les concepts et les techniques présentés ci-dessous :

- Tout d'abord il faut pouvoir accéder aux données stockées au sein des sources de données locales (nous reprenons ici les termes introduits dans le chapitre 1). A l'instar du niveau traduit de l'architecture d'un SGMB, il est donc nécessaire de disposer d'entités virtuelles représentant ces données. Celles-ci seront décrites dans la suite de ce mémoire par le concept de relai (aussi appelé proxy en anglais). Ce concept représente des structures virtuelles, sortes de coquilles vides, identiques aux structures réelles manipulées par le langage mais ne contenant pas de données propres, celles-ci étant conservées au sein des sources locales. Notons que tout comme dans les SGMB, ces entités doivent être structurées suivant le métamodèle du langage. La création de ces entités peut donc devoir faire appel à des outils permettant de convertir les données représentées depuis le métamodèle de leurs sources respectives au métamodèle du langage. Dans la suite de ce mémoire, nous supposerons néanmoins que toutes les données sont représentées suivant le même métamodèle, celui du langage. Nous verrons plus loin qu'il pourra être aisé de spécialiser notre système pour y inclure des convertisseurs.
- L'introduction de relai au sein du langage nous place au niveau traduit de l'architecture des SGMB. En remontant cette architecture, on trouve ensuite le niveau fédéré et le niveau externe. Comme cela a été décrit dans le chapitre 1, ces deux niveaux peuvent être décrits et gérés par un seul et même système de vues. Voici donc le deuxième concept à introduire au sein du langage de programmation. Notons que, à l'inverse du concept de relai, celui de vue peut concerner aussi bien les données persistantes que volatiles. En effet, les premières sont représentées par des relais et les secondes à l'aide des structures réelles du langage. Comme les relais doivent être structurellement identiques à ces dernières, il est très intéressant d'autoriser la définition de vues aussi bien au dessus de relai, au dessus de structures réelles du langage et au dessus d'autres vues. Cela est représenté dans la figure 2.5.
- Les deux derniers concepts à introduire au sein du langage concernent les techniques de gestion des données. En particulier, il s'agit de pouvoir garantir l'accès et la cohérence des

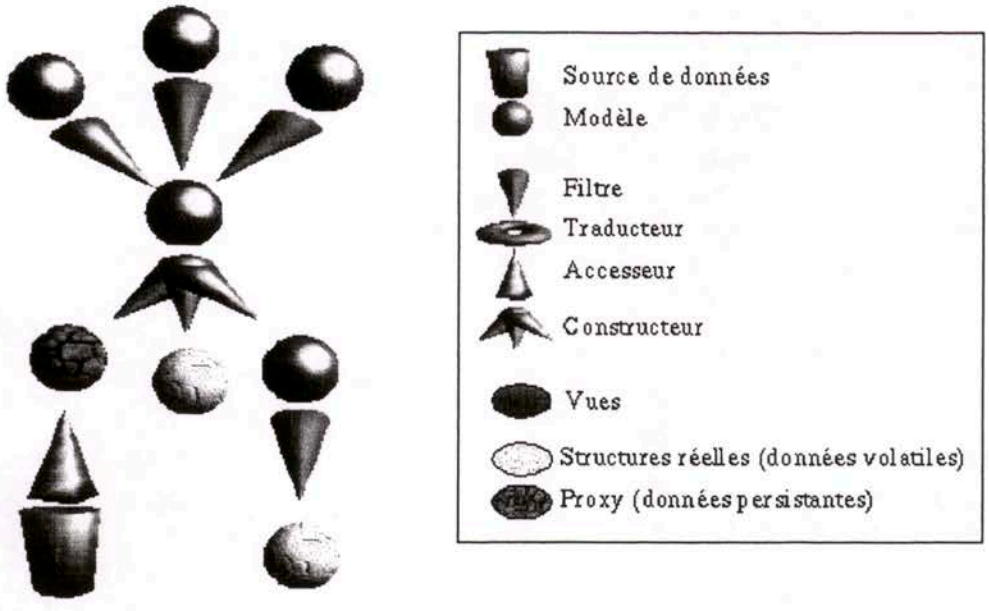


FIG. 2.5 - Architecture modifiée

données. Les concepts de requête et de transaction doivent donc être présents au sein du langage et doivent pouvoir être utilisés, à l'instar du concept de vue aussi bien pour les données volatiles que persistantes.

Pour terminer cette partie, notons que tous les concepts à inclure au sein du langage doivent l'être de manière à homogénéiser l'accès aux sources de donnnées et aux données volatiles. Nous présentons ci-dessous trois principes à respecter dans le processus d'intégration de ces concepts pour garantir cette propriété :

Le principe d'indépendance comportementale. Tous les concepts présentés précédemment doivent pouvoir être mis en œuvre sur des entités du langage quels que soient les comportements appliqués sur ces entités. Par exemple, si l'on considère un modèle à objets et le concept transactionnel, n'importe quelle méthode doit pouvoir faire partie d'une exécution transactionnelle et doit pouvoir être prise en compte pour le contrôle de cohérence.

Le principe d'indépendance de nature. Tous les concepts à l'exception de celui de relai doivent pouvoir être mis en œuvre sur les données persistantes ou volatiles (nous verrons plus loin qu'ils peuvent aussi être aisément appliqués aux données distribuées). Par exemple, il doit pouvoir être possible d'effectuer des requêtes renvoyant des résultats contenant conjointement des entités persistantes et volatiles.

Le principe d'indépendance de type. Tous les concepts définis dans cette partie doivent pouvoir être mis en œuvre sur les entités du langage indépendamment de leur structure. Par exemple, si l'on considère un modèle à objets, il doit être possible de définir une vue à partir de n'importe quelle classe et ceci indépendamment du nombre ou du type de ses champs.

Nous avons décrit les principales propriétés et les principes à respecter quant à l'intégration des concepts de relai, transaction, vue et requête au sein d'un langage de programmation

dans le but de résoudre l'hétérogénéité de stockage et l'hétérogénéité d'accès aux données. Ces propriétés et ces principes nous ont amenés à faire des choix quant à la modélisation de notre système. Nous les décrivons ci-dessous en prenant le concept de transaction comme base de réflexion, puis en étendant celle-ci aux autres concepts précédemment cités lorsque cela est pertinent.

2.2.2 Des Objets

L'interface classique de la majorité des modèles transactionnels est constituée d'opérations, telles que `begin`, `abort` et `commit`, délimitant des blocs d'exécution transactionnelle.

Des modèles transactionnels plus évolués étendent cette interface et y incluent de nouvelles opérations. Par exemple, dans [Kaiser93] sont définies des opérations telles que `split` pour restructurer dynamiquement les transactions ; la mise en œuvre du protocole de validation en deux phases (2PC) (voir [Bernstein et al.87]) requiert une opération de préparation nommée `prepare_to_commit`, etc. De plus, la sémantique d'une même opération peut varier d'un modèle transactionnel à l'autre. Par exemple, l'opération `commit` qui indique la fin d'une transaction et sa validation, n'a pas le même comportement si elle est appelée sur une transaction plate ou sur une sous-transaction dans le contexte du modèle des transactions imbriquées [Moss85] ou sur un des membres d'une transaction coopérante [Fernandez et al.89]. En effet, dans le modèle transactionnel plat classique, la validation d'une transaction implique que ses effets soient atomiques, durables et visibles pour toute nouvelle transaction alors que dans le modèle des transactions imbriquées, la validation d'une sous-transaction a pour conséquence de ne rendre visible les effets de celle-ci qu'à sa transaction parente et aux descendants de cette dernière. Dans le modèle de transactions coopérantes, la validation ne rend visible les effets d'une transaction qu'aux autres transactions du même groupe.

Tout ceci montre le besoin de définir une interface transactionnelle pouvant être étendue par l'ajout de nouvelles opérations et polymorphique pour permettre la redéfinition du comportement des opérations.

Ces deux propriétés sont des propriétés intrinsèques aux langages à objets. Cela nous amène à choisir un tel langage comme base de notre étude et ainsi à introduire les transactions comme des objets, chaque modèle transactionnel pouvant être représenté par une classe. Toute transaction est donc, dans notre modèle, l'instance d'une classe et s'exécute suivant la sémantique définie dans cette classe.

Toute application peut sous-classer une ou plusieurs classes transactionnelles et donc spécialiser le modèle transactionnel utilisé en fonction de ses besoins, de manière dynamique, et ceci tout en conservant une approche transactionnelle générique. De plus, si le langage de programmation utilisé gère le chargement dynamique de classes, toute application peut utiliser de nouveaux modèles transactionnels sans être recompilée, tant que les classes implantant ces systèmes sont conformes aux besoins de l'application en termes de comportement.

Le choix d'un métamodèle à objets pour structurer les entités du langage n'est pas sans conséquence sur l'intégration du concept de relai au sein de celui-ci. Si l'on se place par rapport à l'architecture du SGMB, cela revient à choisir un métamodèle global à objets. Ceci implique de devoir définir des relais structurés sous la forme de classes au dessus de sources de données hétérogènes et donc cela nécessite de disposer d'outils capables de traduire des données

provenant de sources de données hétérogènes en données structurées suivant un métamodèle à objets. Si l'on considère qu'aujourd'hui la majorité des données qu'il est souhaitable de prendre en compte dans un tel outil sont structurées à 70% suivant un métamodèle relationnel, le reste étant réparti entre des métamodèles à objets et des métamodèles semi-structurés, en particulier HTML ou XML, il est donc nécessaire d'avoir à disposition des outils de transformation adaptés à ces métamodèles, et notamment au métamodèle relationnel et aux métamodèles semi-structurés. En ce qui concerne le premier, de nombreuses recherches ont été menées ces dernières années et se sont concrétisées par de nombreux prototypes (voir par exemple [Lebastard93]). Pour les seconds, de nombreuses recherches sont en cours et l'on peut dès à présent disposer d'un certains nombre de prototypes (voir par exemple [Simeon99]).A l'instar de Vodak [Klas et al.96b], IRO-DB [Gardarin95], nous considérons qu'un métamodèle à objets est actuellement l'un des candidats idéal pour l'intégration de sources de données hétérogènes (voir notre discussion sur ce sujet en 1.4.2).

Ce choix influe bien évidemment aussi sur le choix du langage de requêtes et sur le système de vues à mettre en œuvre au sein du langage puisque ceux-ci devront prendre en compte les spécificités des langages à objets et notamment les concepts de méthode, d'héritage et d'objet imbriqué. En particulier, la programmation par objets, à l'inverse par exemple de la programmation fonctionnelle tels que Lisp, nécessite un effort important de description des classes et donc des structures de l'application avant de définir des comportements et des objets. Nous avons décrit en 1.4.2 les différentes approches possibles concernant la mise en œuvre de vues qui sont l'approche descriptive et l'approche transformationnelle. Le concept de vue semble devoir être intégré au sein d'un langage à objets suivant une approche descriptive qui correspond bien à l'esprit du langage plutôt que suivant une approche transformationnelle qui correspond plutôt à l'esprit d'un langage fonctionnel.

2.2.3 Factorisation

Une transaction est selon [Lynch83] :

- une unité logique regroupant les opérations nécessaires à la complétion d'une tâche ;
- une unité cohérente dont l'exécution préserve la cohérence du système d'information ;
- une unité atomique qui assure l'exécution de toutes les opérations qu'elle contient ou d'aucune d'entre elles.

Cette définition est intéressante car elle met en valeur les deux composantes d'une transaction :

Une composante de contrôle de l'exécution : toute transaction est une unité logique qui structure et contrôle l'exécution d'une application, chaque modèle transactionnel offrant des possibilités de structuration différentes. Par exemple, le modèle transactionnel plat n'autorise que la définition de transactions de plus haut niveau. A l'inverse, le modèle des transactions emboîtées permet de définir des structures transactionnelles complexes, imbriquées les unes dans les autres à la manière des poupées russes.

Une composante de gestion des données : l'exécution d'une transaction doit vérifier les propriétés de cohérence et d'atomicité ou, plus généralement les propriétés ACID. Ces propriétés sont bien souvent décrites en termes de lectures et d'écritures de données. Il s'agit, par exemple, de garantir la cohérence des données malgré des écritures et lectures concurrentes ou de pouvoir annuler les écritures effectuées par une transaction

pour garantir l'atomicité de celle-ci. Ces propriétés sont donc intrinsèquement liées aux données.

Il semble intéressant de découpler ces deux composantes dans le but de permettre la spécialisation de l'une ou de l'autre indépendamment. En généralisant ce raisonnement, il nous semble intéressant de découpler de même les propriétés de la composante de gestion de données, chaque objet pouvant vérifier une certaine combinaison de celles-ci, et ce, indépendamment des autres objets. Dans notre système, le concept de transaction représente ainsi une unité structurant et contrôlant l'exécution d'une application, telle que tout objet impliqué dans la transaction soit géré par une combinaison de différentes propriétés. En d'autres termes :

- Tous les objets impliqués dans une même transaction ne sont pas nécessairement gérés de la même manière. Par exemple, l'exécution d'une transaction peut vérifier toutes les propriétés ACID pour les objets persistants, et seulement les propriétés ACI pour les objets distribués.
- Le comportement transactionnel de chaque objet est dynamique c'est-à-dire qu'il peut être modifié lors de l'exécution de l'application. Cette propriété est très importante et permet notamment de créer des bibliothèques de classes dont les instances possédent des propriétés transactionnelles, celles-ci pouvant être définies par l'utilisateur a posteriori et non a priori par le concepteur des classes. Notons que pour éviter des problèmes de cohérence, nous imposons que la modification du comportement transactionnel d'un objet soit effectuée en dehors de toute transaction.
- La composante de contrôle de l'exécution d'une transaction est clairement séparée de la composante de gestion des données. L'utilisateur peut donc spécialiser la composante de contrôle sans pour autant spécialiser les algorithmes de gestion de données et inversement.

Intéressons nous à présent à l'intégration de vues au sein du langage. Dans le cadre d'une approche déclarative telle que présentée en 1.4.2.1 et d'un langage à objets, une vue est avant tout un ensemble d'entités virtuelles équivalentes aux classes(voir par exemple [Lebastard93] et [Radeke et al.95]). Bien évidemment, ces entités sont liées à d'autres classes ou à d'autres entités virtuelles à l'aide desquelles sont calculées les instances de la vue. De telles entités virtuelles peuvent être décrites en deux étapes. La première consiste à décrire la structure interne et le comportement de chaque entité d'une manière identique à la description de la structure interne et du comportement d'une classe. La seconde consiste à décrire les correspondances entre l'entité virtuelle et les classes ou entités virtuelles sur lesquelles elle est basée. Pour les mêmes raisons que celles nous ayant poussé à factoriser le concept de transaction, il nous semble intéressant de découpler ces deux composantes de la description des vues, à savoir la structure et le comportement de l'entité virtuelle que nous nommerons dans la suite la classe virtuelle et les correspondances entre cette structure et les classes de bases que nous nommerons mapping. Dans notre système, le concept de vue est ainsi un ensemble d'entités, chacunes étant l'association d'une structure virtuelle appelée classe virtuelle et d'un ou plusieurs mappings décrivant les correspondances entre la classe virtuelle et des classes de bases spécifiées au sein du mapping. En d'autres termes :

- Une classe virtuelle est décrite de la même manière qu'une classe « normale ». En particulier, et pour respecter les principes d'indépendance définis précédemment, il doit être possible d'y inclure n'importe quel type de champs et n'importe quel type de méthode. De même, il doit être possible que cette classe virtuelle puisse hériter d'autres classes, virtuelles ou non.
- Lorsqu'une classe virtuelle est associée à un seul mapping, l'ensemble de ses instances est l'ensemble des objets résultant du calcul des correspondances décrites au sein du mapping

à partir des instances des classes de base. On dira dans ce cas que l'extension de la classe virtuelle est l'extension du mapping.
- Lorsqu'une classe de base est associée à plusieurs mappings, l'ensemble de ses instances est l'union des extensions de chaque mapping.

A l'instar de la dynamicité du comportement transactionnel, un mapping doit pouvoir être associé à une classe virtuelle (ou supprimé) dynamiquement. Cette propriété est encore plus intéressante si le langage permet de charger dynamiquement de nouveaux mappings. Dans ce cas, il sera par exemple possible d'associer les classes virtuelles existantes de l'application à de nouvelles classes sans stopper l'application. En particulier, il sera possible de prendre en compte et d'intégrer de nouvelles sources de données sans stopper l'application.

2.2.4 Sémantique implicite

La plupart des bibliothèques transactionnelles utilisées lors du développement d'applications intègrent des opérateurs nécessaires à la création, au démarrage, à la validation et à l'annulation de transactions. Bien souvent, elles contiennent aussi les opérateurs nécessaires à la pose de verrou, à la génération des informations de reprise sur panne, au contrôle de concurrence. Bien que ce genre de bibliothèque soit très souple et permette une gestion très fine des transactions, son emploi reste très complexe et implique de « truffer » le code d'appels explicites aux opérateurs transactionnels. En conséquence, les modules ainsi définis sont difficilement portables, la logique de l'application est brouillée, le programmeur ne doit plus seulement se concentrer sur ses algorithmes mais aussi sur la manière de gérer la concurrence ou l'atomicité.

D'autre part, dans la définition de [Lynch83], une transaction est une unité structurant et contrôlant l'exécution d'une application. La particularité des transactions, par rapport aux autres structures de contrôle d'un langage de programmation telles que les blocs d'exécution, est la possibilité d'arrêter l'exécution courante pour éventuellement annuler les effets de la transaction, si l'on se place dans un cadre de gestion atomique. Ce mécanisme est plus ou moins déjà présent au sein des langages à objets :

- L'exécution de l'application est contrôlée et structurée par les envois de messages.
- L'exécution d'une méthode peut être stoppée par le gestionnaire d'exception.

Nous proposons d'utiliser ces propriétés des langages à objets pour gérer les transactions de manière presque transparente. A cet effet, nous introduisons le concept d'*envoi de message transactionnel* tel que les données manipulées dans une méthode exécutée par envoi de message transactionnel soient gérées par une transaction créée automatiquement avant le début de la méthode. Si la méthode termine, la transaction est validée. Si une exception, d'un type particulier, est lancée, la transaction est annulée. L'avantage de notre approche est double :

- Le programmeur n'a pas à effectuer des tâches complexes telles que poser des verrous ou gérer explicitement des mises à jour concurrentes. Le temps de développement des applications et le risque d'ajouter des erreurs de conception dues à l'utilisation d'un système transactionnel sont donc réduits.
- La transformation d'un code non transactionnel en un code transactionnel est triviale. En effet, seuls quelques envois de messages doivent être modifiés en envois de messages transactionnels. Il n'y a pas d'appels explicites à une bibliothèque transactionnelle, ce qui augmente la possibilité de réutilisation du code.

L'intérêt de bénéficier d'une sémantique implicite ne se limite pas au concept de transaction. Intégrer des sources de données hétérogènes, pouvant être accédées par des langages de requêtes hétérogènes nécessite la définition d'un langage de requête global homogène. Nous avons déjà vu que celui-ci devait pouvoir prendre en compte les spécificités d'un métamodèle à objets pour être homogène avec ce dernier. De plus, pour garantir les principes d'indépendance, ce langage de requête doit permettre de manière homogène de rechercher par le contenu des objets instances de classes virtuelles, de relais ou de classes normales. Pour les mêmes raisons que celles nous ayant poussé à définir une sémantique transactionnelle implicite, il semble aussi intéressant de faire en sorte que la mise en œuvre de ce langage de requêtes ne nécessite pas l'apprentissage de nouveaux concepts distincts de ceux inclus dans le langage de programmation. En d'autres termes, alors que l'on aurait pu choisir un langage de requêtes à objets plus ou moins standardisé tels que OQL ou SQL3, il nous a semblé préférable de faire en sorte que les requêtes puissent être exprimées au sein de notre système à l'aide de la même syntaxe et des mêmes concepts que ceux utilisés dans le langage de programmation. Nous verrons plus loin que ce choix implique certaines limitations du pouvoir d'expression du langage de requête. Mais en contrepartie, il ne nécessite aucun apprentissage supplémentaire. De plus, à l'image de ce qui a été dit précédemment, le temps de développement des applications et le risque d'ajouter des erreurs de conception dues à l'utilisation de requêtes sont donc réduits.

2.2.5 Réflexivité

Pour terminer cette partie, intéressons nous plus particulièrement au type de langage susceptible d'accueillir les concepts de transaction, de requête, de vue et de relai. Nous avons vu dans la partie 2.2 quelques règles caractérisant nos choix de modélisation quant à l'intégration de ces concepts au sein d'un langage de programmation « classique ». En particulier, il ressort de cette étude que le langage considéré doit reposer avant tout sur un métamodèle à objets pour d'une part garantir les propriétés de polymorphisme ou de spécialisation nécessaires par exemple à la mise en œuvre d'un système transactionnel paramétrable et évolutif , et d'autre part, bénéficier du fort pouvoir d'expression de ce métamodèle et permettre ainsi l'intégration du plus grand nombre de sources de données. Lors de cette discussion, nous avons aussi été amené à définir certains concepts nouveaux tels que l'envoi de message transactionnel, les mappings ou les classes virtuelles. Ceux-ci ne sont pas originellement présents au sein des langages à objets. L'intégration des concepts de transaction, vue, requête et relai au sein d'un langage peut donc être réalisée, telle que nous la souhaitons, soit par définition d'un nouveau langage et de son métamodèle associé, soit par l'extension d'un langage existant et de son métamodèle associé. Il nous paraît plus judicieux de choisir la seconde approche, car elle permet d'une part de bénéficier des travaux déjà menés sur le langage ou le métamodèle considéré, et d'autre part, de nous concentrer sur les concepts à intégrer à ceux-ci plutôt qu'aux différents problèmes de définition d'un tel langage. Enfin, pour résoudre l'hétérogénéité d'accès, il est nécessaire de laisser la possibilité à l'utilisateur de notre système de spécialiser celui-ci.

Cette approche nécessite que le langage et donc son métamodèle associé soient ouverts et spécialisables. Si l'on considère par exemple, le langage C++ ou plus généralement un langage dont le métamodèle n'est pas spécialisable, il est évident, de part la nature même du langage, qu'il est impossible de lui ajouter les concepts d'envoi de message transactionnel, de mapping ou de classe virtuelle. Une famille de langages ou plus précisément une famille de métamodèles répondent parfaitement à ces besoins d'évolution et d'ouverture. Ces sont les métamodèles à

objets dits réflexifs. (voir [Kickzales et al.91] et [Ferber89]). Nous définissons et détaillons ce type de métamodèle dans la partie suivante.

2.3 Le langage de base

Nous présentons dans cette partie des métamodèles et des langages à objets réflexifs. Cette étude sera l'occasion de définir un modèle formel général correspondant à un métamodèle à objets réflexif. Une formalisation restreinte a l'aspect fonctionnel des métamodèles réflexifs ($\lambda-calcul$) à fait l'objet de quelques études (voir par exemple [Mendhekar et al.93]). Nous nous intéresserons dans ce mémoire à l'aspect structurel des langages réflexifs. Une formalisation basée sur cet aspect n'a, à notre connaissance, encore jamais été entreprise pour ce type de métamodèle et constitue donc une part de nos contributions. A l'instar de ce qui à déjà été dit en 1.4, une telle formalisation permet notamment de présenter différents concepts ou techniques indépendamment du langage choisi pour les implanter, et ainsi de se concentrer sur les difficultés de modélisation plutôt que sur les difficultés de réalisation associées au langage considéré. Nous utiliserons cette formalisation pour décrire notre système dans les chapitres suivants.

2.3.1 Les langages réflexifs à objets

2.3.1.1 Faiblesses des langages à objets non réflexifs

La notion de « boîte noire » Les logiciels implantés à l'aide d'un langage à objets sont construits en respectant la règle suivante qui est une implication directe du concept d'envoi de message : par son interface, chaque module explicite ses fonctionnalités mais cache son implantation. Ce principe, informellement nommé « boîte noire » (voir [Kickzales et al.91]), permet principalement de porter, de réutiliser et de maintenir facilement les programmes ou les bibliothèques.

Cette approche de la programmation n'est cependant pas acceptable dans tous les cas de figure. En effet, il n'est pas toujours possible de programmer un module, une bibliothèque ou un logiciel sans connaître le mode d'exploitation de celui-ci par l'utilisateur final. En d'autres termes, l'utilisateur est parfois plus à même de choisir la bonne stratégie d'implantation que le concepteur du logiciel, ou du module, comme nous le verrons dans l'exemple étudié à la fin de cette partie. Or, d'après le principe de la « boîte noire » , l'utilisateur ne peut pas modifier l'implantation du module qui est « enfermé dans la boîte ».

Prenons un exemple, considérons un système de gestion de fichiers sur ordinateur. Le concepteur de ce système veut écrire un programme efficace et rapide. Il peut alors choisir la stratégie suivante : subdiviser le disque dur en entités de taille égale et inscrire chaque fichier au début d'une nouvelle entité. Ainsi, plus la taille des entités sera grande, plus la recherche des fichiers sera rapide. En contrepartie, plus la taille des entités sera importante et plus l'espace disque sera restreint (les entités ne seront pas remplies au maximum). En conséquence, le concepteur du système de gestion de fichiers doit faire un choix sur la taille minimale. Le système sera alors très efficace si l'utilisateur manipule de gros fichiers (relativement à la taille minimale choisie). Mais si ce dernier travaille en majorité sur de petits fichiers, il perdra rapidement

beaucoup d'espace disque. Cet exemple met en lumière qu'un conflit est inévitable dans le cas où le programmeur choisit une stratégie qui est en désaccord avec celle de l'utilisateur. Notons que ceci rejoint le problème soulevé en 2.1.1 induit par la nécessité pour l'admistrateur d'un SGMB fédéré simple de choisir a priori la taille et le contenu du modèle de celui-ci sans connaître les besoins de l'utilisateur.

Non-uniformité du métamodèle En général, les entités définies dans le cadre d'un métamodèle à objets non réflexifs n'ont pas toutes le même statut. On peut les répartir en deux catégories :

- les classes, génératrices d'instances ;
- les instances terminales.

Dans les modèles non réflexifs, les classes ne sont pas des objets à part entière car elles ne sont pas elles-mêmes instances de classes. Elles ne peuvent ni envoyer de messages, ni en recevoir. Par ailleur,les instances ne peuvent pas créer d'autres instances. Le programmeur utilisant un langage à objets non réflexif doit donc faire face à une certaine incohérence, utiliser l'envoi de message pour la communication entre les objets mais utiliser une approche procédurale pour la création des objets c'est-à-dire pour l'instanciation, les classes n'étant pas elles-même des objets.

2.3.1.2 Définitions

Pour résoudre les deux problèmes précédemment soulevés, il est nécessaire d'une part de transformer notamment les classes en objets et ainsi uniformiser le langage, et d'autre part, de rendre le langage et son métamodèle spécialisables pour permettre à l'utilisateur de résoudre les éventuels conflits.

Pour que les classes soient des objets, il suffit qu'elles soient instances d'autres classes, appelées métaclasses. Notons que pour ne pas avoir déplacé le problème correspondant au niveau des métaclasses, il est nécessaire que celles-ci soient instances d'autres métaclasses et ainsi de suite, créant une hiérarchie d'instanciation. Dans certains systèmes à métaclasses, on peut avoir ainsi une régression infinie. Si l'on considère qu'il est préférable de stopper cette régression, il faut alors définir un circuit dans le graphe d'instanciation, soit définir un ensemble de classes formant un circuit, à partir duquel il est possible de trouver tous les objets du langage. Souvent ce circuit est de dimension un Pour ce faire, le système contient une métaclasse instance d'elle-même et racine de la hiérarchie d'instanciation. Dans un système à métaclasses, toutes les entités sont des objets et sont instances d'une classe. Les objets ne pouvant pas avoir d'instances et correspondant aux instances « classiques » sont dits instances terminales.

L'introduction des métaclasses dans un système à objets résoud le problème d'uniformité du langage. Elle peut aussi permettre de le rendre spécialisable. En effet, introduire des métaclasses revient à définir le comportement des classes, à l'aide de méthodes. Si les concepts d'héritage, d'envoi de message, de typage, ... sont définis de la sorte et que les métaclasses peuvent être spécialisées par exemple par héritage, il devient alors possible à l'utilisateur de modifier la sémantique du langage. Les langages mettant en œuvre ces propriétés sont dits réflexifs. Les objets du langage nécessaires à la gestion et au contrôle des objets, comme les

classes ou les métaclasses, sont appelés métaobjets. Enfin, si la marche à suivre pour spécialiser un langage réflexif à métaobjets est définie sous la forme de protocoles, on parlera alors de protocoles à métaobjets.

Avant de présenter un modèle formel d'un tel langage, nous présentons dans la suite quelques langages interprétés réflexifs à objets parmi les plus connus. Les langages compilés mettant en œuvre un protocole à métaobjets nécessitent quelques adaptations des concepts introduits ici et seront présentés ultérieurement.

2.3.1.3 Smalltalk-76

Les langages Smalltalk font certainement partie des précurseurs en termes d'objets et de protocoles à métaobjets. C'est en fait dans Smalltalk-76 (voir [Ingalls78]) qu'apparaît pour la première fois (tous langages confondus) la notion de métaclasse. Smalltalk fait partie des langages qui ont choisi de ne pas avoir de régression infinie. Une métaclasse unique, nommée `Class` est ainsi instance d'elle-même, et tout autre classe ou métaclasse, est instance directement ou indirectement de `Class`. Smalltalk-76 contient aussi une racine d'héritage unique appelée `Object`.

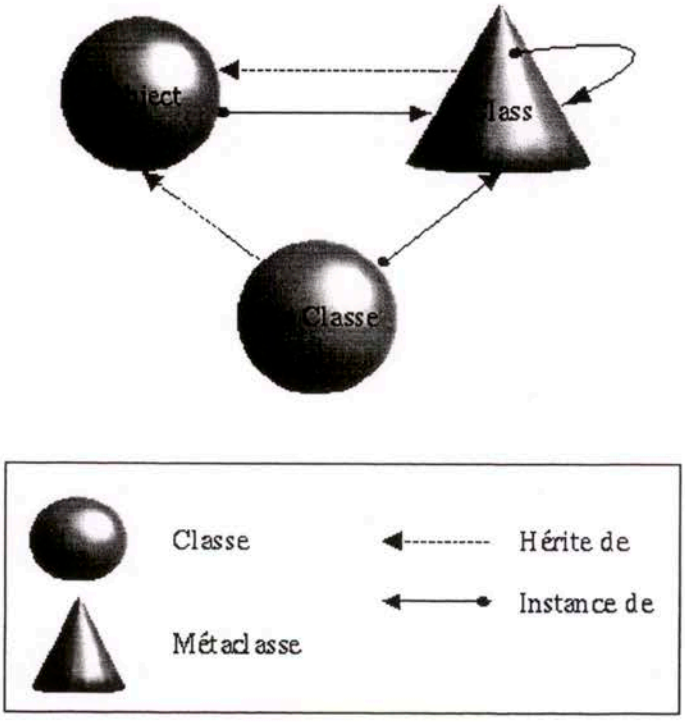


FIG. 2.6 – Smalltalk-76

En conséquence, toute classe ou métaclasse, en particulier `Class`, hérite, directement ou indirectement, d'`Object`. De plus, toute classe ou toute métaclasse, en particulier `Object` est instance, directement ou indirectement, de `Class`. Ceci est représenté dans la figure 2.6.

La restriction majeure de Smalltalk-76 consiste en l'impossibilité de créer de nouvelles métaclasses et donc de spécialiser le langage et sa sémantique.

2.3.1.4 Smalltalk-80

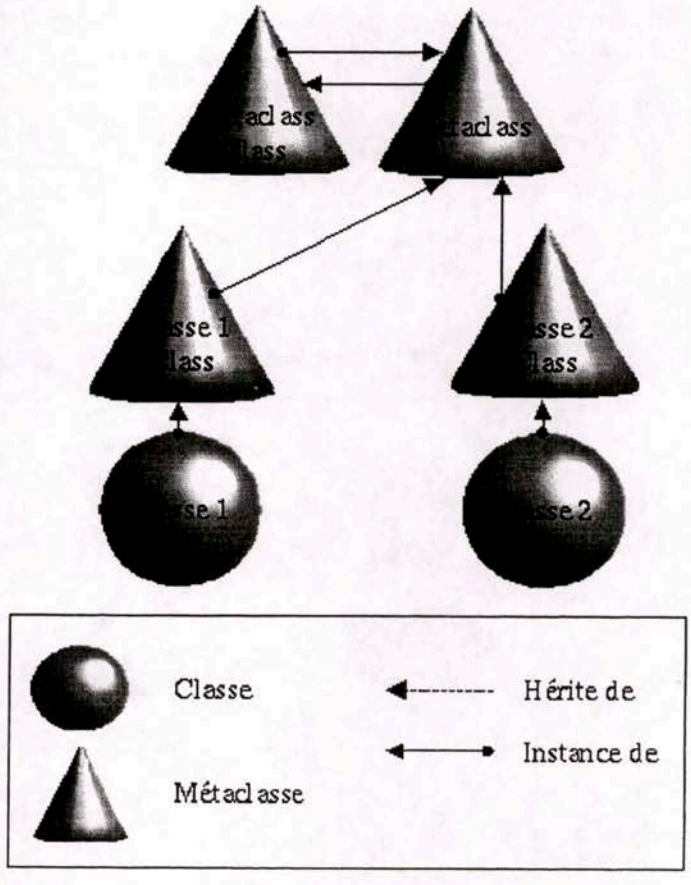


FIG. 2.7 – Smalltalk-80

Smalltalk-80 [Goldberg et al.83] reprend les principes de base introduits dans les versions précédentes, à savoir :

- Toutes les entités du langage sont des objets
- Tout objet est instance d'une classe ou d'une métaclasse qui définit sa structure et son comportement.
- Un objet est uniquement accessible par envoi de message.

La différence principale entre le métamodèle de Smalltalk-80 et celui de Smalltalk-76 réside en la généralisation des métaclasses. Dans Smalltalk-80, il existe non plus une seule métaclasse, mais une métaclasse distincte pour chaque classe. Toutes les métaclasses sont instances d'une seule méta-métaclasse appelée `Metaclass`, elle même étant instance de `Metaclass class` qui est instance de `Metaclass`. Notons que le circuit d'instanciation permettant d'éviter une régression infinie est ici de dimension 2. Ce métamodèle est ainsi constitué de 5 niveaux : les

instances terminales, les classes, les métaclasses, `Metaclass` et `Metaclass class` (voir la figure 2.7).

Dans ce langage, chaque classe est instance d'une métaclasse distincte, créée automatiquement par le système. Cette architecture, comme celle de Smalltalk-76 rend le langage uniforme. A l'inverse de Smalltalk-76 qui ne contenait qu'une seule métaclasse ce qui impliquait de ne pouvoir spécialiser le comportement des classes et donc du langage, l'utilisateur peut, lors de la description des classes, modifier le comportement de celles-ci, en précisant comment leurs métaclasses respectives doivent être créées. Smalltalk-80 est donc un véritable langage réflexif spécialisable.

2.3.1.5 ObjVLisp

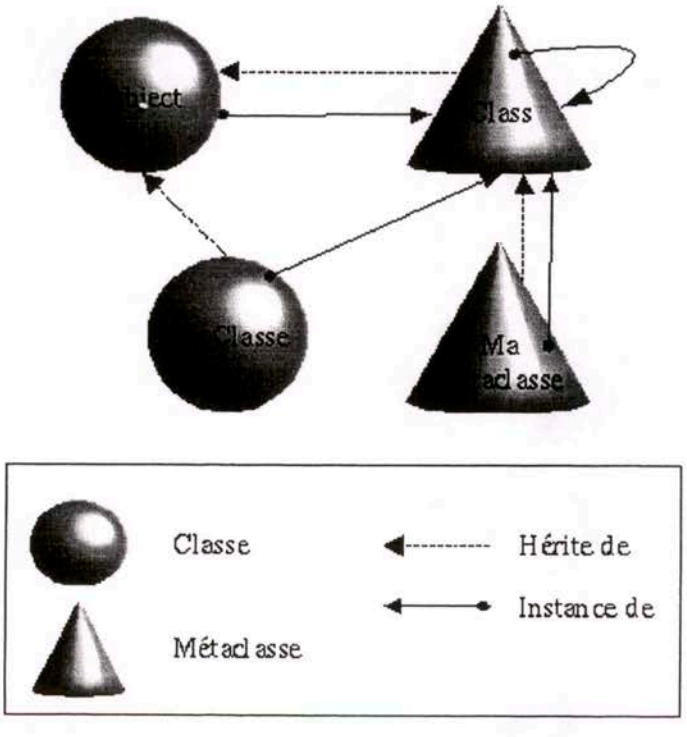


FIG. 2.8 – ObjVLisp

ObjVLisp [Cointe87] est souvent considéré comme l'un des langages réflexifs les plus intéressants du point de vue de l'uniformisation et des capacités de spécialisation du langage. Comme Smalltalk-76, ce langage est basé sur deux métaobjets (voir 2.8) :

- `Class`, racine du graphe d'instanciation
- `Object`, racine du graphe d'héritage.

Cependant, à l'inverse de Smalltalk-76, il est possible de définir de nouvelles métaclasses de manière homogène à la définition de nouvelles classes, rendant ainsi le langage et ses comportements complètement spécialisables.

Enfin, ce langage, pouvant être implanté en Scheme, un dialecte LISP minimal, est complètement construit et défini à partir de `Class` et `d'Object`, ce qui en fait un langage complètement réflexif, simple et minimal.

2.3.1.6 CLOS

CLOS (Common Lisp Object System) (voir [Bobrow et al.88], [Paepcke90] et [Attardi et al.89]) se veut avant tout une extension objet de Common Lisp, le standard américain des dialectes LISP. Ce langage est construit suivant deux niveaux : l'interface de programmation et le protocole à métaobjets.

- L'interface de programmation fournit les primitives pour créer et manipuler les objets. Cette interface se subdivise elle-même en deux sous-parties :

  1. une interface fonctionnelle qui contient les macros implantant les opérations de base.
  2. une interface de macros qui permet à l'utilisateur de définir rapidement classes et méthodes.

- Le protocole à métaobjets est assez similaire à celui d'ObjVLisp dans le sens où CLOS est complètement réflexif et permet de créer des métaclasses à volonté.

Alors qu'ObjVLisp est simple, on repproche souvent à CLOS sa complexité. J.C. royer a émis de nombreuses critiques à ce propos sur ce langage (voir [Royer92]) concernant en particulier un manque d'homogénéité de sa syntaxe ou l'introduction de concepts marginaux rendant difficile notamment l'utilisation du protocole à métaobjets.

2.3.2 Un modèle Formel

Dans cette partie nous présentons un modèle formel basé sur celui décrit en 1.4. Ce modèle est adapté à la représentation mathématique de métamodèles à objets réflexifs tels que celui d'ObjVLisp, langage à partir duquel nous structurerons notre étude.

Nous pouvons dès à présent soulever l'une des principales différences entre le modèle formel d'O2 et celui que nous souhaitons développer. Comme toute entité d'un langage réflexif doit être un objet, les métaclasses, les classes et les instances terminales appartiennent à un même ensemble : l'ensemble des objets. Dans un langage à objets classique, il est impossible de considérer les classes et les objets comme entités d'un même ensemble. Comme nous l'avons précédemment présenté, cette distinction se traduit par la mise en œuvre de deux concepts distincts : celui de modèle correspondant à l'ensemble des classes et celui de base ou d'instance de ce modèle. Si l'on supprime la distinction entre classe et objet, on supprime donc aussi la distinction entre modèle et base. En généralisant cette réflexion, on peut aussi supprimer la distinction entre modèle et métamodèle, ce dernier étant dans un système réflexif à objets composé d'objets.

Cette abscence de distinction entre métamodèle, modèle et instance de celui-ci peut se caractériser par le concept d'état instantané du langage. On peut définir l'état instantané du langage comme l'ensemble des objets, c'est-à-dire à l'ensemble des métaclasses, des classes et des instances terminales, mis en œuvre à un instant donné au sein du langage. Cet ensemble

d'objets représente à la fois le comportement, les données et les structures du langage et de l'application. Il évolue lors de la création ou la destruction d'objets, c'est-à-dire lors de l'évolution du graphe d'instanciation. Ce graphe est donc une bonne représentation du langage à un instant donné. Nous commencerons donc la description de notre modèle formel par la présentation du formalisme d'instanciation.

Nous verrons ensuite comment ajouter à cela, les concepts d'héritage, de type et de comportement qui permettent de déterminer la structure et le comportement des objets.

Enfin, nous verrons comment adapter à notre modèle formel les règles de compatibilité de métaclasses (voir [Bouraquadi et al.96]) qui permettent de « garantir » la cohérence du langage et des applications.

2.3.2.1 Instanciation

Le modèle d'instanciation décrit précédemment pour le métamodèle à objets d'O2 est basé sur la fonction de peuplement présentée par l'équation 1.14. Il est souhaitable de mettre en place une fonction similaire dans notre modèle formel. Pour ce faire, nous suivrons les étapes suivantes :

- Tout d'abord nous décrirons l'ensemble des identifiants d'objet sur lequel sera basé la fonction de peuplement. Notons qu'il n'est pas nécessaire de définir ici un ensemble des noms de classes, car celles-ci peuvent être représentées par un identifiant d'objet.
- Puis nous définirons une fonction d'instanciation associant à chaque objet sa classe. Nous avons vu que dans un métamodèle réflexif à objets tel que celui que nous souhaitons modéliser, i.e. un métamodèle contenant une racine d'instanciation unique et tel que chaque objet ne soit instance que d'une seule classe, l'ensemble des objets forme une hiérarchie d'instanciation. Pour définir cette hiérarchie, il est plus naturel et plus précis de définir une fonction associant à chaque objet sa classe, plutôt que la fonction inverse, soit la fonction de peuplement, associant à chaque classe ou métaclasse l'ensemble de ses instances.
- Comme nous venons de le souligner, l'ensemble des objets du langage forme une hiérarchie d'instanciation, il est donc souhaitable comme cela a été fait pour la hiérarchie d'héritage du métamodèle d'O2, de définir une relation sur cette hiérarchie, dite relation d'instanciation. Cette relation sera définie à partir de la fonction d'instanciation.
- Enfin, nous terminerons cette partie par la définition de la fonction de peuplement, plus naturelle du point de vue du programmeur que la fonction d'instanciation.

Ensembles Nous appellerons $\mathcal{O}$ l'ensemble infini dénombrable des identifiants d'objet. Soit O inclus dans $\mathcal{O}$, définissons les identifiants d'objet représentant respectivement la racine d'héritage (`Object` dans ObjVLisp) et la racine d'instanciation (`Class` dans ObjVLisp) comme deux éléments de O : o_{Object} et o_{Class}.

Fonction d'instanciation Nous appellerons fonction d'instanciation sur O, une fonction γ de O dans O telle que si l'on considère γ^n la composée de γ avec elle-même n fois, telle que $\gamma^0 = Id$, Id étant la fonction identité, et $\gamma^n = \gamma \circ \gamma^{n-1}$, on ait alors :

- o_{Class} est un point fixe de la fonction γ, soit $\gamma(o_{Class}) = o_{Class}$.

- o_{Class} est un point fixe de la suite γ^n, soit pour tout $o \in O$, il existe un entier relatif $k > 0$ tel que $\gamma^k(o) = o_{Class}$.

La première propriété stipule que `Class` est instance d'elle-même. La seconde propriété indique que chaque objet doit être instance directement ou indirectement de `Class`. On souhaite en effet que ce métaobjet soit racine d'instanciation.

Objet vide Nous appellerons o_{Void} l'instance de o_{Object} représentant l'absence de valeur, tel que $\gamma(o_{Void}) = o_{Object}$.

Relation d'instanciation Nous appellerons relation d'instanciation sur O, notée $\dashv$, la relation associée à γ telle que pour tout $a, b \in O$, $a \dashv b$ si et seulement si il existe un entier relatif $k \geq 0$ tel que $b = \gamma^k(a)$.

Fonction de peuplement Nous appellerons fonction de peuplement sur O, notée π,une fonction de O dans l'ensemble des parties finies de O, noté $P^{fin}(O)$, telle que pour tout $o \in O$, $\pi(o) = \{o' \in O \mid \gamma(o') = o\}$.

En résumé :

$$\gamma : \left| \begin{array}{c} O \rightarrow O \\ o \mapsto \gamma(o) \mid (\gamma(o_{Class}) = o_{Class}) \wedge (\forall o \in O, \exists k > 0, \gamma^k(o) = o_{Class}) \end{array} \right. \tag{2.1}$$

$$\forall a, b \in O : a \dashv b \Leftrightarrow \exists k \geq 0 \mid b = \gamma^k(a) \tag{2.2}$$

$$\pi : \left| \begin{array}{c} O \rightarrow P^{fin}(O) \\ o \mapsto \pi(o) = \{o' \in O \mid \gamma(o') = o\} \end{array} \right. \tag{2.3}$$

2.3.2.2 Propriétés du modèle d'instanciation

Dans cette partie, nous présentons quelques propriétés intéressantes du modèle d'instanciation introduit précédemment, en particulier nous montrons que :

1. la relation d'instanciation est une relation d'ordre partielle.
2. la hiérarchie d'instanciation telle que définie par la fonction d'instanciation est un pseudo-arbre. Nous appellerons pseudo-arbre un graphe ne contenant qu'un seul circuit de dimension 1 situé sur un objet racine du graphe. Notons qu'il est impossible que la fonction d'instanciation puisse définir un arbre car il existe par définition un circuit de dimension 1 sur o_{Class}. Une conséquence de cette propriété est que l'ensemble des objets du langage peut être déterminé par la connaissance de la racine du pseudo-arbre et de la fonction de peuplement. Notons que dans la description du métamodèle d'O2, l'ensemble des objets était déterminé par un couple formé de l'ensemble des classes et de la fonction de peuplement. Bien que plus général, un métamodèle réflexif nécessite donc moins d'informations pour déterminer l'ensemble des objets.

Commençons par déterminer quelques propriétés simples des fonctions et relations décrites précédemment.

- Tout d'abord, il est évident que o_{Class} est l'unique point fixe de la fonction γ. En effet, soit $o \in O$ point fixe de la fonction γ, on a $\gamma(o) = o$. Or, par définition de γ, $\forall o \in O$, il existe un entier relatif $k > 0$ tel que $\gamma^k(o) = o_{Class}$. Il vient donc $o = o_{Class}$.

$$\forall o \in O, \gamma(o) = o \Leftrightarrow o = o_{Class} \tag{2.4}$$

- Il est aussi clair que, comme $\forall o \in O$, il existe un entier relatif $k > 0$ tel que $\gamma^k(o) = o_{Class}$, tout élément de O est en relation d'instanciation avec o_{Class} .

$$\forall o \in O, o \dashv o_{Class} \tag{2.5}$$

$\dashv$ est une relation d'ordre partielle. Pour montrer cette propriété, il faut montrer que $\dashv$ est réflexive, transitive et anti-symétrique. Notons que $\dashv$ ne peut être une relation d'ordre totale car deux éléments de O ne sont pas forcément en relation.

- $\dashv$ est réflexive : soit $o \in O$ et soit $k = 0$, on a bien $o = \gamma^k(o) = o$ et donc $o \dashv o$.
- $\dashv$ est transitive : soient $a, b, c \in O$ tels que $a \dashv b \dashv c$. Par définition de $\dashv$, il existe deux entiers relatifs $k_1 \geq 0$ et $k_2 \geq 0$ tels que $b = \gamma^{k_1}(a)$ et $c = \gamma^{k_2}(b)$. Il vient $c = \gamma^{k_1+k_2}(a)$ et donc $a \dashv c$.
- $\dashv$ est anti-symétrique : soient $a, b \in O$ tels que $a \dashv b \dashv a$, montrons que $a = b$. Pour ce faire, procédons par l'absurde. Si $a \neq b$, il existe deux entiers relatifs $k_1 > 0$ et $k_2 > 0$ tels que $b = \gamma^{k_1}(a)$ (1) et $a = \gamma^{k_2}(b)$ (2). Si $a = o_{Class}$, (1) implique que $b = o_{Class} = a$, ce qui est impossible. Dans le cas contraire, il existe, en combinant (1) et (2) $m = k_1 + k_2 > 0$ tel que $a = \gamma^{k_1+k_2}(a) = \gamma^m(a)$ (3). De plus, d'après 2.5, il existe un entier relatif $q > 0$ tel que $o_{Class} = \gamma^q(a)$ (4). Si l'on considère p et s, respecivement le quotient et le reste de la division euclidienne de q par m, on a $q = m * p + s$ (5) avec $s < m$. D'après (3) et (4), il vient $o_{Class} = \gamma^{m*p+s}(a) = \gamma^s(a)$ et donc $a = \gamma^{m-s}(o_{Class}) = o_{Class}$, ce qui est impossible.

La relation d'instanciation définit un pseudo-arbre Pour montrer cette propriété, considérons la suite π_n définie de la manière suivante :

$$\pi_n : \left| \begin{array}{l} \pi_0 = \pi \\ \forall k > 0, \forall o \in O, \pi_k(o) = \pi_{k-1}(o) \cup \left(\bigcup_{o' \in \pi_{k-1}(o)} \pi(o') \right) \end{array} \right. \tag{2.6}$$

Le graphe d'instanciation peut être modélisé par cette suite de fonctions, puisqu'elle permet de connaître pour chaque objet l'ensemble de ses instances, des instances de celles-ci, etc.., soit le sous-graphe d'instanciation correspondant à cet objet. Pour prouver que le graphe d'instanciation est un pseudo-arbre, il faut montrer d'une part qu'il a une racine unique et d'autre part qu'il ne possède aucun circuit à l'exception de celui défini sur sa racine. La première partie de la propriété correspond à montrer qu'il existe un objet permettant de générer tous les autres, soit qu'il existe $o \in O$ tel que $\lim_{n\to\infty} \pi_n(o) = O$. Notons qu'il est très probable que cet objet soit o_{Class}. Nous chercherons ainsi à montrer que $\lim_{n\to\infty} \pi_n(o_{Class}) = O$. La deuxième propriété correspond à montrer que pour tout $o \in O$ différent de l'objet racine,

la limite de la suite π_n appliquée à o ne contient pas o. En estimant toujours que o_{Class} soit cette racine, nous prouverons que $\forall o \neq o_{Class} \in O, o \notin \lim_{n\to\infty} \pi_n(o)$.

Pour ce faire, nous allons calculer pour tout $o \in O$, la limite de la suite π_n. Notons que cette limite existe. En effet, la suite converge pout tout $o \in O$ car elle est croissante par définition et bornée par O.

Pour faciliter ce calcul, commençons par prouver les deux propriétés suivantes :

1. $\forall n \geq 0, \forall o \in O : \{o' \in O \mid o = \gamma^{n+1}(o')\} \subseteq \pi_n(o)$
2. $\forall n > 0, \forall o \neq o_{Class} \in O : \{o' \in O \mid o = \gamma^{n+1}(o')\} = \pi_n(o) \setminus \pi_{n-1}(o)$

La première propriété se prouve par récurrence :

- Pour $n = 0$: Par définition de la suite π_n, on a $\forall o \in O, \pi_0(o) = \pi(o) = \{o' \in O \mid o = \gamma(o')\}$
- Supposons la propriété vraie au rang $n-1$: Soit $o \in O$ et soit $o' \in O$ tel que $o = \gamma^{n+1}(o')$, on a $o = \gamma^n(\gamma(o'))$. Par hypothèse de récurrence, il vient $o'' = \gamma(o') \in \pi_{n-1}(o)$. Donc comme $o' \in \pi(o'')$, $o' \in \bigcup_{o'' \in \pi_{n-1}(o)} \pi(o'')$, et ainsi $o' \in \pi_n(o)$.

La seconde propriété se prouve elle aussi par récurrence :

- Pour $n = 1$: D'après la définition de la suite π_n, on a $\forall o \in O, \pi_1(o) \setminus \pi_0(o) \subseteq \bigcup_{o' \in \pi_0(o)} \pi(o')$. Or, par définition de la fonction π, $\forall o \in O, \pi(o) = \{o' \in O \mid o = \gamma(o')\}$. Donc $\forall o \in O, \pi_1(o) \setminus \pi_0(o) \subseteq \bigcup_{o' \in \pi_0(o)} \{o'' \in O \mid o' = \gamma(o'')\}$. De plus, on a $\forall o' \in \pi(o) : o = \gamma(o')$. Il vient donc $\forall o \in O, \pi_1(o) \setminus \pi_0(o) \subseteq \{o'' \in O \mid o = \gamma^2(o'')\}$. Pour prouver l'inclusion inverse, soit $o \neq o_{Class} \in O$, considérons $o'', o' \in O$ tel que $o = \gamma^2(o'')$ et $o' = \gamma(o'')$. On a évidemment $o = \gamma(o')$ et ainsi, par définition de la fonction π, $o' \in \pi(o)$ et $o'' \in \pi(o')$. Il vient donc que $o'' \in \bigcup_{o' \in \pi_0(o)} \pi(o')$. De plus, si $o \neq o_{Class}$, $o'' \notin \pi_0(o)$. En effet, dans le cas contraire, $\gamma(o'') = o = o' = \gamma(o)$ et donc $o = o_{Class}$, ce qui est impossible par hypothèse. En conséquence $o'' \in \pi_1(o) \setminus \pi_0(o)$, ce qui prouve la propriété au rang 1.
- Supposons la propriété vraie jusqu'au rang $n-1$: D'après la définition de la suite π_n, on a $\forall o \in O, \pi_n(o) \setminus \pi_{n-1}(o) = (\bigcup_{o' \in \pi_{n-1}(o) \setminus \pi_{n-2}(o)} \pi(o') \cup \bigcup_{o' \in \pi_{n-2}(o)} \pi(o')) \setminus \pi_{n-1}(o)$.Or, toujours par définition de définition la suite π_n, $\forall o \in O, \bigcup_{o' \in \pi_{n-2}(o)} \pi(o') \subseteq \pi_{n-1}(o)$. Il vient donc $\forall o \in O, \pi_n(o) \setminus \pi_{n-1}(o) \subseteq \bigcup_{o' \in \pi_{n-1}(o) \setminus \pi_{n-2}(o)} \pi(o')$. Par hypothèse de récurrence, $\forall o \neq o_{Class} \in O : \{o' \in O \mid o = \gamma^n(o')\} = \pi_{n-1}(o) \setminus \pi_{n-2}(o)$, d'où $\forall o \neq o_{Class} \in O$, $\pi_n(o) \setminus \pi_{n-1}(o) \subseteq \bigcup_{\{o' \in O \mid \gamma^n(o') = o\}} \pi(o') = \bigcup_{\{o' \in O \mid \gamma^n(o') = o\}} \{o'' \in O \mid \gamma(o'') = o'\}$. En conséquence, il vient $\forall o \neq o_{Class} \in O$, $\pi_n(o) \setminus \pi_{n-1}(o) \subseteq \{o' \in O \mid o = \gamma^{n+1}(o')\}$. Pour prouver l'inclusion inverse, considérons $o \neq o_{Class} \in O$ et $o' \in O$ tels que $\gamma^{n+1}(o') = o$. Soit $o'' \in O$ tel que $o'' = \gamma(o')$, on a donc $o' \in \pi(o'')$ et $o = \gamma^n(o'')$. Par hypothèse de récurence, $o'' \in \pi_{n-1}(o) \setminus \pi_{n-2}(o)$. Par définition de la suite π_n, il vient donc $o' \in \pi_n(o)$. Si $o' \in \pi_{n-1}(o)$, $o' \in (\pi_{n-1}(o) \setminus \pi_{n-2}(o)) \cup ... \cup (\pi_1(o) \setminus \pi_0(o)) \cup \pi_0(o)$. Par hypothèse de récurence, il existe un entier relatif k tel que $n \geq k \geq 1$ et $\gamma^k(o') = o$. Il vient donc $o = \gamma^{n+1-k}(o) = \gamma^m(o)$ avec $m > 0$ et donc $o = o_{Class}$ ce qui est impossible. On a donc bien $o' \in \pi_n(o) \setminus \pi_{n-1}(o)$, ce qui conclue cette preuve.

A l'aide des deux propriétés précédemment décrites, il est désormais possible de calculer la limite de la suite π_n. Pour ce faire, dissocions le cas $o = o_{Class}$ et $o \neq o_{Class}$.

- Si $o = o_{Class}$, on a, d'après la propriété 1, $\forall n \in \mathbb{N} : \{o' \in O \mid o_{Class} = \gamma^{n+1}(o')\} \subseteq \pi_n(o_{Class})$. En réalisant l'union sur n de chaque côté de l'inclusion, on obtient l'inclusion suivante : $\forall k \in \mathbb{N}$, $\bigcup_{n=0}^{k} \{o' \in O \mid o_{Class} = \gamma^{n+1}(o')\} \subseteq \bigcup_{n=0}^{k} \pi_n(o_{Class})$. Or, comme

nous l'avons précédemment souligné, la suite π_n est croissante. En conséquence, $\forall k \in \mathbb{N}, \bigcup_{n=0}^{k} \{o' \in O \mid o_{Class} = \gamma^{n+1}(o')\} \subseteq \pi_k(o_{Class})$. En passant à la limite de chaque côté, on obtient finalement $\lim_{n\to\infty} \pi_n(o_{Class}) \supseteq \bigcup_{\mathbb{N}} \{o' \in O \mid o_{Class} = \gamma^{n+1}(o')\} = O$. Or, comme nous l'avons précédemment souligné, $\lim_{n\to\infty} \pi_n(o_{Class})$ est bornée par O. On a bien ainsi $\lim_{n\to\infty} \pi_n(o_{Class}) = O$.
- Si $o \neq o_{Class}$, on a, d'après la propriété 2 $\forall n \in \mathbb{N}^* : \{o' \in O \mid o = \gamma^{n+1}(o')\} = \pi_n(o) \setminus \pi_{n-1}(o)$. Or, $\forall n \in \mathbb{N}^*, \pi_n(o) = \bigcup_{k=1}^{n} (\pi_k(o) \setminus \pi_{k-1}(o)) \cup \pi_0(o)$. On obtient ainsi $\forall n \in \mathbb{N}^*, \pi_n(o) = \bigcup_{k=1}^{n} \left\{o' \in O \mid o = \gamma^{k+1}(o')\right\} \cup \pi_0(o) = \bigcup_{k=1}^{n+1} \left\{o' \in O \mid o = \gamma^{k}(o')\right\}$. En passant à la limite, il vient finalement $\forall o \neq o_{Class} \in O : \lim_{n\to\infty} \pi_n(o) = \{o' \in O \mid o' \dashv o\} \setminus \{o\}$.

En résumé :

- $\lim_{n\to\infty} \pi_n(o_{Class}) = O$
- $\forall o \neq o_{Class} \in O : \lim_{n\to\infty} \pi_n(o) = \{o' \in O \mid o' \dashv o\} \setminus \{o\}$

Ces deux égalités prouvent d'une part que o_{Class} est racine du graphe d'instanciation, et que d'autre part, celui-ci est un pseudo-arbre car comme la limite de la suite π_n appliquée à tout $o \neq o_{Class}$ ne contient pas o, le graphe ne contient qu'un seul circuit sur o_{Class}.

2.3.2.3 Héritage

Le concept d'héritage est un concept tout aussi important que celui d'instanciation. Il garantit les propriétés d'évolution et de spécialisation du langage et des applications ainsi que la capacité de développement modulaire qui caractérise les langages à objets. Il est certain que sans le concept d'héritage, ceux-ci n'auraient pris autant d'ampleur.

Dans les langages à objets, ce concept se traduit par des liens entre classes. Comme nous l'avons déjà présenté, ce lien est souvent associé à une sémantique de spécialisation. Par exemple, si l'on considère une classe V représentant des véhicules, il doit être possible, par héritage, de créer une classe plus précise que V, par exemple celle représentant des véhicules à moteurs, en ne spécifiant que les différences par rapport à V. L'héritage permet ainsi de décomposer le langage en plusieurs modules indépendants mais sémantiquement reliés faciliant ainsi leur développement et leur maintenance.

Il est souhaitable de mettre en œuvre un lien d'héritage au sein de notre modèle formel. Nous supposerons que ce lien est un lien d'héritage multiple, soit qu'une classe peut hériter simultanément de plusieurs autres classes. Pour ce faire, nous suivrons les étapes suivantes :

- Pour être homogène à la description de l'instanciation qui se compose d'une fonction de peuplement associant à chaque classe ou métaclasse l'ensemble de ses instances, nous définirons une fonction de sous-classement associant à chaque classe l'ensemble de ses sous-classes.
- Comme précédemment, nous déduirons une relation d'ordre partielle de cette fonction : la relation d'héritage. Notons que cette relation est l'équivalent de celle décrite pour le modèle d'O2.
- Enfin, nous terminerons cette partie par la définition de la fonction d'héritage, plus naturelle du point de vue du programmeur que la fonction de sous-classement.

Fonction de sous-classement Soit β une fonction de O dans $P^{fin}(O)$, l'ensemble des parties finies de O et soit β^n la suite des images successives de β définie comme suit :

$$\begin{array}{l} \beta^0 : \left| \begin{array}{l} O \rightarrow P^{fin}(O) \\ o \mapsto \{o\} \end{array} \right. \\ \forall k > 0, \beta^k : \left| \begin{array}{l} O \rightarrow P^{fin}(O) \\ o \mapsto \bigcup_{o' \in \beta^{k-1}(o)} \beta(o') \end{array} \right. \end{array} \tag{2.7}$$

La fonction β sera appelée fonction de sous-classement si et seulement si elle vérifie les propriétés suivantes :

- $o_{Class} \in \beta(o_{Object})$
- $\forall o \in O, \forall n > 0, o \notin \beta^n(o)$
- $\bigcup_{k \in \mathbb{N}} \beta^k(o_{Object}) = (\bigcup_{o \in O} \beta(o)) \cup \{o_{Object}\}$

La première propriété spécifie que `Class` hérite `d'Object`. (nous reprenons ici la sémantique du métamodèle d'ObjVLisp). La seconde propriété indique que le graphe d'héritage doit être sans circuit. Enfin, la dernière précise que `Object` doit être la racine d'héritage. Notons que les deux dernières propriétés impliquent que le graphe d'héritage est un arbre de racine `Object`.

Relation d'héritage Nous appellerons relation d'héritage sur O, notée $\prec$, la relation associée à β telle que pour tout $a, b \in O$, $a \prec b$ si et seulement si il existe un entier relatif $k \geq 0$ tel que $a \in \beta^k(b)$.

Fonction d'héritage Nous appellerons fonction d'héritage sur O, notée ϖ, une fonction de O dans $P^{fin}(O)$, telle que pour tout $o \in O$, $\varpi(o) = \{o' \in O \mid o \in \beta(o')\}$.

En résumé

$$\beta : \left| \begin{array}{l} O \rightarrow P^{fin}(O) \\ o \mapsto \beta(o) : \\ \quad o_{Class} \in \beta(o_{Object}) \\ \quad \forall o \in O, \forall k > 0, o \notin \beta^k(o) \\ \quad \bigcup_{k \in \mathbb{N}} \beta^k(o_{Object}) = (\bigcup_{o \in O} \beta(o)) \cup \{o_{Object}\} \end{array} \right. \tag{2.8}$$

$$\forall a, b \in O : a \prec b \Leftrightarrow \exists k \geq 0, a \in \beta^k(b) \tag{2.9}$$

$$\varpi : \left| \begin{array}{l} O \rightarrow P^{fin}(O) \\ o \mapsto \{o' \in O \mid o \in \beta(o')\} \end{array} \right. \tag{2.10}$$

2.3.2.4 Propriétés du modèle d'héritage

Comme précédemment, nous allons essayer de déterminer les propriétés intéressantes de notre modèle d'héritage.

Tout d'abord, notons qu'il était impossible de spécifier l'ensemble des classes et des métaclasses à l'aide du seul modèle d'instanciation. En effet, comme il peut exister des classes ou des métaclasses sans instance, par exemple les classes abstraites en C++, la fonction d'instanciation ne permet pas de séparer les instances terminales des classes ou des métaclasses.

L'ensemble des classes et des métaclasses peut néanmoins être déterminé à l'aide de la fonction d'héritage, ou plus précisément, à l'aide de o_{Object} et de la fonction d'héritage. Nous avons en effet vu que cet objet était la racine unique de l'arbre d'héritage, toutes les classes ou métaclasses étant dès lors des sous-classes, directes ou indirectes, d'`Object`. Si l'on appelle $\mathcal{M}_o$ l'ensemble des classes et des métaclasses, celui-ci peut-être obtenu par l'équation suivante :

$$\mathcal{M}_o = \bigcup_{k \in \mathbb{N}} \beta^k(o_{Object}) \tag{2.11}$$

Notons aussi que, comme on pouvait le prévoir, tout élément de $\mathcal{M}_o$ est en relation d'héritage avec `Object`. En effet, par définition $\forall o \in \mathcal{M}_o, \exists k \in \mathbb{N} \mid o \in \beta^k(o_{Object})$. Donc, il est évident que $\forall o \in \mathcal{M}_o, o \prec o_{Object}$.

De la même manière, on peut déterminer l'ensemble des métaclasses, noté $\mathcal{M}_c$, comme l'ensemble des objets héritant de `Class`. Il en résulte que $\mathcal{M}_c$ peut être obtenu par l'équation suivante :

$$\mathcal{M}_c = \bigcup_{k \in \mathbb{N}} \beta^k(o_{Class}) \tag{2.12}$$

La différence entre $\mathcal{M}_o$ et $\mathcal{M}_c$ caractérise l'ensemble des classes, noté $\mathcal{C}$.

$$\mathcal{C} = \mathcal{M}_o \setminus \mathcal{M}_c \tag{2.13}$$

Enfin, la différence entre O et $\mathcal{M}_o$ représente l'ensemble des instances terminales, noté $\mathcal{I}$.

$$\mathcal{I} = O \setminus \mathcal{M}_o \tag{2.14}$$

A l'instar de ce que nous avons présenté précédemment pour la relation d'instanciation, montrons que la relation d'héritage est une relation d'ordre partielle sur O.

$\prec$ est une relation d'ordre partielle. Notons que $\prec$ ne peut être une relation d'ordre totale car deux éléments de O ne sont pas forcément en relation.

Pour montrer cette propriété, commençons par prouver l'implication suivante :

$$\forall o_1, o_2 \in O : \exists m \in \mathbb{N}, o_2 \in \beta^m(o_1) \Rightarrow \forall n \in \mathbb{N}, \beta^n(o_2) \subseteq \beta^{n+m}(o_1) \tag{2.15}$$

Cette implication est évidemment vraie pour $n = 0$. Supposons qu'elle soit vraie au rang $n-1$ et prouvons la au rang n. Soit donc $o_1, o_2 \in O : \exists m \in \mathbb{N}, o_2 \in \beta^m(o_1)$, et soit $o \in \beta^n(o_2)$. Par définition de la suite β^n, il existe $o' \in \beta^{n-1}(o_2)$ tel que $o \in \beta(o')$. Or, par hypothèse de récurrence, $o' \in \beta^{n-1+m}(o_1)$. Il vient finalement, $o \in \beta^{n+m}(o_1)$.

Il reste à montrer que $\prec$ est réflexive, transitive et anti-symétrique

- $\prec$ est réflexive car pour tout $o \in O$, $o \in \beta^0(o) = \{o\}$ et donc $o \prec o$.
- $\prec$ est transitive. Soient $o_1, o_2, o_3 \in O$ tels que $o_1 \prec o_2$ et $o_2 \prec o_3$. Par définition de la relation $\prec$, $\exists k_1, k_2 \in I\!N$ tels que $o_1 \in \beta^{k_1}(o_2)$ et $o_2 \in \beta^{k_2}(o_3)$. A l'aide de l'implication 2.15, on a $\beta^{k_1}(o_2) \subseteq \beta^{k_1+k_2}(o_3)$. En conséquence, il vient $o_1 \in \beta^{k_1+k_2}(o_3)$ et donc $o_1 \prec o_3$.
- $\prec$ est anti-symétrique. Soient $o_1, o_2 \in O$ tels que $o_1 \prec o_2$ et $o_2 \prec o_1$.Par définition de la relation $\prec$, $\exists k_1, k_2 \in I\!N$ tels que $o_1 \in \beta^{k_1}(o_2)$ et $o_2 \in \beta^{k_2}(o_1)$. A l'aide de l'implication 2.15, on a $\beta^{k_1}(o_2) \subseteq \beta^{k_1+k_2}(o_1)$. En conséquence, il vient $o_1 \in \beta^{k_1+k_2}(o_1)$. Donc, par définition de la suite β^n, $k_1 + k_2 = 0$ ce qui implique, $k_1 = k_2 = 0$, $k_1$et k_2 étant éléments de $I\!N$. On a finalement $o_1 \in \{o_2\}$ et $o_2 \in \{o_1\}$ et donc $o_1 = o_2$.

Extension Enfin, comme nous l'avions fait lors de la description du modèle formel d'O2, introduisons le concept d'extension au sein de notre formalisation. L'extension d'une classe ou d'une métaclasse représente l'ensemble de ses instances directes et des instances de ses sous-classes directes ou indirectes. Notons qu'il ne faut pas confondre l'extension d'une classe ou d'une métaclasse m avec l'ensemble des objets éléments de l'arbre d'instanciation de racine m. Nous supposerons en étendant cette notion que o_{Void} fait partie de l'extension de n'importe quelle classe ou métaclasse.

Considérons la fonction $\widetilde{\pi}$ associant à chaque classe ou métaclasse $o \in \mathcal{M}_o$ son extension. Cette fonction est définie de la manière suivante :

$$\widetilde{\pi} : \left| \begin{array}{l} \mathcal{M}_o \rightarrow P^{fin}(O) \\ o \mapsto \{o_{Void}\} \cup \{\bigcup_{o' \prec o} \pi(o')\} \end{array} \right. \tag{2.16}$$

2.3.2.5 Typage

Jusqu'à présent, tous les ensembles manipulés étaient des ensembles d'objets. Afin de définir le type des objets, il est nécessaire, à l'instar de ce qui a été défini en 1.4.1.6, de considérer l'ensemble des types. Comme nous souhaitons modéliser un métamodèle réflexif, il est tentant de représenter les types comme des objets. Pour ce faire, considérons deux objets identifiés par o_{Type} et $o_{MetaType}$ tels que :

- $o_{MetaType} \prec o_{Class}$ et $o_{MetaType} \dashv o_{Class}$
- $o_{Type} \prec o_{Object}$ et $o_{Type} \dashv o_{MetaType}$
- $\forall o \in \widetilde{\pi}(o_{MetaType}) \setminus \{o_{Void}\} : o \prec o_{Type}$

Nous appellerons type tout objet référencé par $o \in \mathcal{M}_o$ tel que $o \in \widetilde{\pi}(o_{MetaType})$. L'ensemble des référents de type, ou pour simplifier ensemble des types, noté $\mathcal{T}$, est donc défini par l'équation suivante :

$$\mathcal{T} = \{o \in O \mid o \in \widetilde{\pi}(o_{MetaType})\} \tag{2.17}$$

Attachons-nous désormais à définir le type d'un objet. Comme nous considérons les types comme des objets, cela peut être décrit simplement par une fonction associant à chaque objet son type. Cependant, le type d'un objet est un concept complexe mettant en œuvre à la fois les paradigmes d'instanciation et d'héritage. Par exemple, si l'on considère un objet instance d'une classe A, celle-ci héritant de deux autres classes B et C, le type de l'objet sera le

résultat d'une opération complexe effectuée sur les types définis dans chacune de ces classes. Cette assertion amène plusieurs remarques :

- Tous les objets d'une même classe ont le même type.
- Le type d'un objet est défini par le biais de sa classe et des superclasses de celles-ci.
- Comme nous nous plaçons dans un contexte réflexif, l'opération de calcul du type d'un objet peut-être décrit comme une méthode du langage et peut donc être redéfinie et/ou spécialisée.
- Le type d'un objet ne peut donc être décrit comme une simple fonction associant à chaque objet un autre objet.

Pour définir le type d'un objet, commençons par distinguer deux sortes de type :

- Le type intrinsèque de l'objet, soit celui défini dans sa classe. Dans notre modélisation, ce type est déterminé à l'aide d'une fonction de $\mathcal{M}_o$ dans $\mathcal{T}$, nommée σ_i, associant à chaque classes ou métaclasse le type intrinsèque de ses instances, telle que $\sigma_i(o_{Object}) = o_{Void}$.
- Le type hérité de l'objet soit celui résultant des types définis dans les superclasses de sa classe. Dans notre modélisation, ce type est déterminé à l'aide d'une fonction de $\mathcal{M}_o$ dans $\mathcal{T}$, nommée σ_h, associant à chaque classe ou métaclasse le type hérité de ses instances, telle que $\sigma_h(o_{Class}) = \sigma_h(o_{Object}) = \sigma_i(o_{Object}) = o_{Void}$.

Cette séparation doit nous permettre d'expliciter la possibilité de spécialiser le calcul du type d'un objet en fonction d'un type intrinsèque et d'un type hérité. Notons aussi que nous ne souhaitons pas, par soucis de clarté de l'exposé et de notre modélisation, expliciter la possibilité de spécialiser le calcul du type hérité. Nous supposons en effet ici qu'il existe une fonction permettant de calculer ce type à partir des types des superclasses. Une implantation particulière de notre modélisation pourra néanmoins permettre la spécialisation de cette fonction par l'utilisateur.

De plus, définissons une fonction de combinaison f sur $\mathcal{T}$ comme une fonction de $\mathcal{T} \times \mathcal{T}$ dans $\mathcal{T}$ telle que $\forall o_1, o_2 \in \mathcal{T}, f(o_1, o_2) = f(o_2, o_1)$ et $\forall o \in \mathcal{T}, f(o, o_{Void}) = o$. Considérons $\mathcal{X}(\mathcal{T})$, l'ensemble des fonctions de combinaisons sur $\mathcal{T}$. Enfin, soit $\chi_{\mathcal{T}}$ une fonction de $\mathcal{M}_o$ dans $\mathcal{X}(\mathcal{T})$, associant à chaque classe ou métaclasse une fonction de combinaison sur $\mathcal{T}$.

Il est maintenant possible de définir le type d'un objet à l'aide de la fonction σ, de $\mathcal{M}_o$ dans $\mathcal{T}$ associant à chaque classe ou métaclasse le type de ses instances et telle que $\forall o \in \mathcal{M}_o, \sigma(o) = \chi_{\mathcal{T}}(o)(\sigma_i(o), \sigma_h(o))$. En conséquence, le type d'un objet o élément de O, noté $\sigma_o(o)$ est égal à $\sigma(\gamma(o))$.

Notons qu'une propriété évidente de la fonction σ est :

$$\sigma_o(o_{Class}) = \sigma_o(o_{Object}) = \sigma_i(o_{Class}) \tag{2.18}$$

En effet, on a $\sigma_o(o_{Class}) = \sigma_o(o_{Object}) = \sigma(o_{Class})$. Or, d'après la définition de σ, $\sigma(o_{Class}) = \chi_{\mathcal{T}}(o_{Class})(\sigma_i(o_{Class}), \sigma_h(o_{Class}))$. En conséquence, on obtient l'égalité suivante : $\sigma(o_{Class}) = \chi_{\mathcal{T}}(o_{Class})(\sigma_i(o_{Class}), o_{Void})) = \sigma_i(o_{Class})$.

2.3.2.6 Comportements

La modélisation du comportement, i.e. l'ensemble des méthodes, d'un objet est assez similaire à celle du type. En effet, comme pour ce dernier, le comportement d'un objet est obtenu d'une

part, à partir de celui défini au sein de sa classe, et d'autre part, à partir de ceux définis dans les superclasses de celle-ci.

Considérons deux objets identifiés par o_{Method} et $o_{MetaType}$ tels que :

- $o_{MetaMethod} \prec o_{Class}$ et $o_{MetaMethod} \dashv o_{Class}$
- $o_{Method} \prec o_{Object}$ et $o_{Method} \dashv o_{MetaMethod}$
- $\forall o \in \widetilde{\pi}(o_{MetaMethod}) \setminus \{o_{Void}\} : o \prec o_{Method}$

Nous appellerons méthode, tout objet référencé par $o \in O$ tel que et $o \in \widetilde{\pi}(o_{MetaMethod})$. L'ensemble des référents de méthodes, ou pour simplifier l'ensemble des méthodes, noté $\mathcal{B}$, est donc défini par l'équation suivante :

$$\mathcal{B} = \{o \in O \mid o \in \widetilde{\pi}(o_{MetaMethod})\} \tag{2.19}$$

Comme précédemment, distinguons deux sortes de comportement :

- Le comportement intrinsèque de l'objet, soit celui défini dans sa classe. Dans notre modélisation ce comportement est déterminé à l'aide d'une fonction, nommée μ_i, de $\mathcal{M}_o$ dans l'ensemble des parties finies de $\mathcal{B}$ associant à chaque classe ou métaclasse m le comportement intrinsèque de ses instances soit l'ensemble des méthodes définies dans m, telle que $\mu_i(o_{Object}) = \emptyset$.
- Le comportement hérité de l'objet soit celui résultant des comportements définis dans les superclasses de sa classe. Dans notre modélisation, ce comportement est déterminé à l'aide d'une fonction , nommée μ_h, de $\mathcal{M}_o$ dans l'ensemble des parties finies de $\mathcal{B}$ associant à chaque classe ou métaclasse le comportement hérité de ses instances, telle que $\mu_h(o_{Class}) = \mu_h(o_{Object}) = \mu_i(o_{Object}) = \emptyset$.

De plus, définissons une fonction de combinaison f sur $P^{fin}(\mathcal{B})$ comme une fonction de $P^{fin}(\mathcal{B}) \times P^{fin}(\mathcal{B})$ dans $P^{fin}(\mathcal{B})$ telle que :

- $\forall a, b \in P^{fin}(\mathcal{B}), f(a,b) = f(b,a)$;
- et $\forall a \in P^{fin}(\mathcal{B}), f(a,\emptyset) = a$.

Considérons $\mathcal{X}(P^{fin}\mathcal{B})$, l'ensemble des fonctions de combinaison sur $P^{fin}(\mathcal{B})$. Enfin, soit $\chi_{P^{fin}(\mathcal{B})}$ une fonction de $\mathcal{M}_o$ dans $\mathcal{X}(P^{fin}(\mathcal{B}))$, associant à chaque classe ou métaclasse une fonction de combinaison sur $P^{fin}(\mathcal{B})$.

Il est maintenant possible de définir le comportement d'un objet à l'aide de la fonction μ, de $\mathcal{M}_o$ dans $P^{fin}(\mathcal{B})$ associant à chaque classe ou métaclasse le comportement de ses instances et telle que $\forall o \in \mathcal{M}_o, \mu(o) = \chi_{P^{fin}(\mathcal{B})}(o)(\mu_i(o), \mu_h(o))$. En conséquence, le comportement d'un objet o élément de O, noté $\mu_o(o)$ est égal à $\mu(\gamma(o))$.

Notons qu'une propriété évidente de la fonction μ est :

$$\mu_o(o_{Class}) = \mu_o(o_{Object}) = \mu_i(o_{Class}) \tag{2.20}$$

En effet, on a $\mu_o(o_{Class}) = \mu_o(o_{Object}) = \mu(o_{Class})$. Or d'après la définition de μ, $\mu(o_{Class}) = \chi_{P^{fin}(\mathcal{B})}(o_{Class})(\mu_i(o_{Class}), \mu_h(o_{Class}))$. En conséquence, on obtient l'égalité suivante : $\mu(o_{Class}) = \chi_{P^{fin}(\mathcal{B})}(o_{Class})(\mu_i(o_{Class}), \emptyset)) = \mu_i(o_{Class})$.

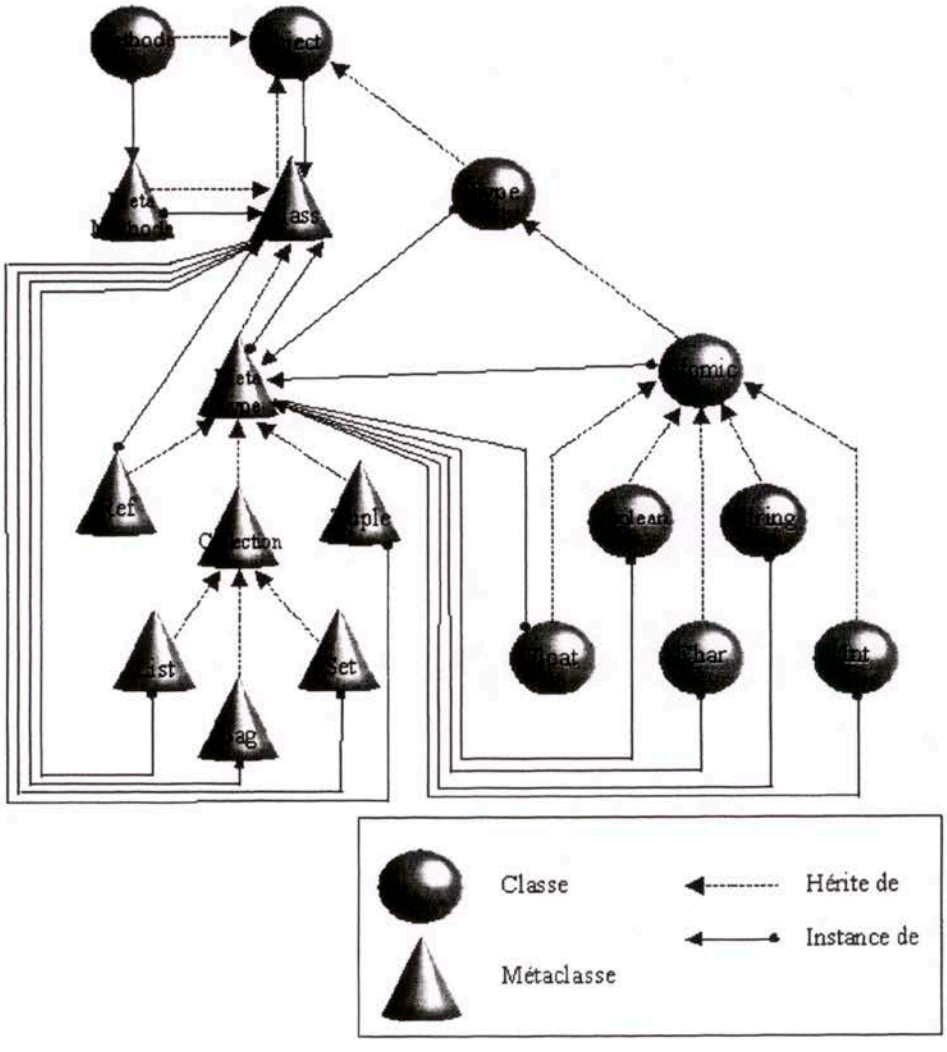


FIG. 2.9 – Hiérarchies d'objets

2.3.2.7 Hiérarchies d'objets

Avant de continuer plus avant dans la description de notre modèle formel, prenons un exemple d'ensemble d'objets vérifiant l'ensemble des propriétés décrites précédemment et pouvant servir à la réalisation d'un véritable langage réflexif à objets.

Nous avons déjà présenté certains objets devant prendre place dans cet ensemble puisque contenus dans notre modèle formel. Il s'agit tout d'abord des objets référencés par les identifiants o_{Class} et o_{Object} à partir desquels tous les autres objets sont construits. Puis, nous avons introduit les objets référencés par o_{Type} et $o_{MetaType}$ permettant de construire l'ensemble des types. Enfin, nous avons présenté les objets référencés par o_{Method} et $o_{MetaMethod}$ permettant la mise place du concept de méthode au sein de notre modèle. Ces six objets doivent être la base de tout langage dérivé de notre formalisation.

L'un des objectifs principaux d'un langage de programmation est de fournir un moyen de re-

présenter les données du monde réel. Pour ce faire, il est nécessaire de structurer ces données à l'aide des types du langage. Dans notre modèle, les types sont aussi des objets. Pour représenter les données du monde réel, il est donc nécessaire de disposer d'une part d'objets types capables de représenter les types atomiques (entier, réel, chaîne de caractères,...) et, d'autre part, d'autres objets type permettant de combiner les types atomiques en types complexes. Nous avons choisi de présenter les objets type suivants (voir figure 2.9).

Les types atomiques. Considérons les types charactère, entier, réel, booléen et chaîne de caractères. Nous identifierons ces types au sein de notre modèle respectivement par o_{Char}, o_{float}, $o_{boolean}$ et o_{String}. Notons que, pour que ces objets soient considérés comme des types, il est nécessaire que $\forall o \in \{o_{Char}, o_{float}, o_{boolean}, o_{String}\}$:

- $o \prec o_{Type}$
- et $o \dashv o_{MetaType}$.

Ces types sont des types atomiques. Pour respecter le paradigme de modularité de la programmation à objets, nous définissons un nouvel objet o_{Atomic} factorisant les propriétés et comportements communs aux types atomiques. Les différents objets présentés ici doivent vérifier les propriétés suivantes :

- $\forall o \in \{o_{Char}, o_{float}, o_{boolean}, o_{String}\}, o \prec o_{Atomic}$
- $o_{Atomic} \prec o_{Type}$
- $\forall o \in \{o_{Atomic}, o_{Char}, o_{float}, o_{boolean}, o_{String}\}, o \dashv o_{MetaType}$

Notons, pour être complet, que ces types, et plus particulièrement o_{Atomic} forment le lien avec l'architecture matérielle de l'ordinateur sur lequel est utilisé le langage. En conséquence, ces objets ne peuvent être décrits en termes d'autres objets du langage, mais à l'aide d'un autre langage, pouvant être le langage machine ! On atteint ici les limites de la réflexivité du langage. Nous supposerons donc que ces objets existent au sein du langage mais ne peuvent être décrits par le langage lui-même.

Les types référence Considérons un objet o_{Ref} permettant de construire un type référençant d'autres objets. Deux approches sont possibles. La première, dite « non typée », consiste à considérer qu'une instance de o_{Ref} référence n'importe quel objet du langage. La seconde, dite « typée », nécessite d'associer un objet à chaque instance de o_{Ref}, permettant de restreindre l'ensemble des objets pouvant être référencés par ces instances aux instances de l'objet associé. Dans un langage qui contient une hiérarchie d'héritage unique, la seconde approche est plus générale car elle contient la première. En effet, il est possible de référencer l'ensemble des objets du langage en utilisant l'instance du type référence associée à la racine d'héritage. En conséquence, nous choisissons la seconde approche. La première approche nous aurait conduit à considérer o_{Ref} comme une classe instance de $o_{MetaType}$ et héritant de o_{Type}. Bien que la seconde approche puisse nous conduire à la même définition de o_{Ref} il nous semble plus judicieux de considérer o_{Ref} comme une métaclasse, instance de o_{Class} et héritant de $o_{MetaType}$. En effet, de cette manière, il est possible de construire, pour chaque classe ou métaclasse m, une classe, instance de o_{Ref} et héritant de o_{Type}, associée à m et représentant le type référence sur les instances m.

Les types collection Alors que la modèle formel d'O2 permettait la construction d'ensembles, nous souhaitons permettre la construction d'ensembles, de listes et de sacs. Pour les mêmes raisons que précédemment, nous choisissons une approche typée. Pour ce faire, nous introduisons quatre nouvelles métaclasses identifiées par $o_{Collection}$, o_{set}, o_{List}, o_{Bag}, telles que :

- $\forall o \in \{o_{Set}, o_{List}, o_{Bag}\}, o \prec o_{Collection}$
- $o_{Collection} \prec o_{MetaType}$
- $\forall o \in \{o_{Collection}, o_{Set}, o_{List}, o_{Bag}\}, o \dashv o_{Class}$

Ainsi, il est possible pour chaque classe ou métaclasse m de construire des classes, instance de o_{Set}, de o_{List} et de o_{Bag}, associées à m et représentant les types respectivement ensemble d'instances de m, liste d'instances de m et sac d'instances de m.

Les n-uplets Pour terminer attachons nous à introduire la notion de n-uplet au sein de notre formalisme. Comme précédemment, nous souhaitons adopter une approche typée. Pour ce faire, introduisons la métaclasse o_{Tuple} telle que :

- $o_{Tuple} \prec o_{MetaType}$
- $o_{Tuple} \dashv o_{Class}$

Toute classe, instance de o_{Tuple}, représente une structure de type n-uplet. Nous supposerons que les méthodes associées à o_{Tuple} permettent de préciser la structure de chacun de ces n-uplets.

Notons enfin que dans notre formalisme le type d'une classe est un type quelconque et peut ainsi être différent d'un n-uplet.

2.3.2.8 Envoi de message

Attachons nous maintenant à modéliser l'envoi de message qui est l'un des concepts les plus importants d'un langage à objets. L'envoi de message est le medium unique de communication entre les objets. Il correspond à une demande d'exécution d'une méthode sur un objet. Cette demande nécessite d'identifier les méthodes pouvant être appliquées sur chaque objet ainsi que les paramètres de ces méthodes. Dans notre formalisme, les méthodes sont des objets et peuvent ainsi être spécifiées par leur identifiant. Les paramètres peuvent, quant à eux, être identifiés par l'ordre suivant lequel ils apparaissent au sein de l'envoi de message. Enfin, il est possible de suivre deux approches distinctes pour formaliser l'envoi de message. Comme précédemment, il s'agit des approches typées et non typées. Comme nous avons choisi une approche typée pour décrire les types et les comportements, il semble judicieux de continuer dans cette voie en ce qui concerne l'envoi de message. Pour formaliser ce concept, nous commencerons par formaliser plus avant les méthodes notamment en précisant leurs signatures et leurs types de retour. Puis, comme nous suivons une approche typée, nous formaliserons la vérification du typage des arguments. Enfin nous présenterons la notion de fonction d'exécution modélisant l'appel d'une méthode sur un objet.

fonction de signature Nous appellerons fonction de signature, notée sig, une fonction de $\mathcal{B}$ dans l'ensemble des listes finies d'éléments de $\mathcal{M}_o$, notée $L^{fin}(\mathcal{M}_o)$ associant à chaque

méthode de $\mathcal{B}$ la liste des classes de ses arguments, telle que $sig(o_{Method}) = ()$, $()$ dénotant la liste vide.

$$sig : \left| \begin{array}{c} \mathcal{B} \rightarrow L^{fin}(\mathcal{M}_o) \\ m \mapsto (o_1, o_2, ..., o_n) : sig(o_{Method}) = () \end{array} \right. \tag{2.21}$$

fonction de retour Nous appellerons fonction de retour, notée ret, une fonction de $\mathcal{B}$ dans $\mathcal{M}_o$ associant à chaque méthode la classe de son résultat telle que $ret(o_{Method}) = o_{Object}$.

$$ret : \left| \begin{array}{c} \mathcal{B} \rightarrow \mathcal{M}_o \\ m \mapsto o : ret(o_{Method}) = o_{Object} \end{array} \right. \tag{2.22}$$

relation de typage Nous appellerons relation de typage, notée $\ll$, une relation sur $L^{fin}(O)$ telle que pour tout $a = (o_1, ..., o_m)$ et $b = (o'_1, ..., o'_n)$, $a \ll b$ si et seulement si $n = m$ et $\forall i \in (1..n), o_i \in \widetilde{\pi}(o'_i)$.

fonction d'exécution Nous appellerons fonction d'exécution, une fonction, notée λ, de $O \times \mathcal{B} \times L^{fin}(O)$ dans O telle que pour tout $o \in O$, pour tout $m \in \mu_o(o)$, pour tout $a \in L^{fin}(O)$ tel que $a \ll sig(m)$, il existe un unique $o' \in \widetilde{\pi}(ret(m))$ tel que $\lambda(o, m, a) = o'$.

Polymorphisme Nous avons décrit la fonction d'exécution formalisant l'envoi de message en utilisant la possibilité d'identifier les méthodes de manière identique à un objet. Cette modélisation à un défaut car elle ne permet pas d'exprimer le concept de polymorphisme, c'est-à-dire la capacité d'envoyer un même message à des objets de classes différente, résultant en des comportements différents. Pour résoudre ce problème, il faut ne plus associer un envoi de message à l'exécution d'une méthode identifiée de façon unique mais l'associer par exemple à un nom, une chaîne de caractères, pouvant correspondre à des méthodes différentes au sein de classes distinctes. Pour ce faire, considérons l'ensemble $\mathcal{S}$ des chaînes de caractères définit comme suit :

$$\mathcal{S} = \widetilde{\pi}(o_{String}) \tag{2.23}$$

Considérons l'ensemble des fonctions de $\mathcal{S}$ dans $\mathcal{B}$, noté $\mathcal{F}$. Nous appellerons fonction d'association une fonction, notée α, de $\mathcal{M}_o$ dans $\mathcal{F}$ tel que $\forall o \in \mathcal{M}_o, \forall s \in \mathcal{S}, \alpha(o)(s) \in \mu(o) \cup \{o_{Void}\}$.

Enfin, définissons l'ensemble des messages possibles sur $o \in \mathcal{M}_o$ comme l'ensemble suivant :

$$\rho(o) = \{s \in \mathcal{S} \mid \alpha(o)(s) \neq o_{Void}\} \tag{2.24}$$

Nous appellerons ρ la fonction de $\mathcal{M}_o$ dans $P^{fin}(\mathcal{S})$ associant à chaque classe ou métaclasse o, l'ensemble des messages possibles sur o.

Fonction d'envoi de message Nous appellerons fonction d'envoi de message une fonction, notée λ', de $O \times \mathcal{S} \times L^{fin}(O)$ dans O telle que pour tout $o \in O$, pour toute chaîne de caractères $s \in \rho(\gamma(o))$, pour tout $a \in L^{fin}(O)$ tel que $a \ll sig(\alpha(\gamma(o))(s))$, il existe un unique $o' \in \widetilde{\pi}(ret(\alpha(\gamma(o))(s))$ tel que $\lambda'(o, s, a) = o' = \lambda(o, \alpha(\gamma(o))(s), a)$.

2.3.2.9 Compatibilité de classes

Pour garantir la propriété de spécialisation du langage, nous n'avons pour l'instant posé que peu de contraintes sur le formalisme présenté. Dans l'absolu, cela peut conduire à des incompatibilités induites par l'héritage. En effet, supposons qu'une classe A soit typée de telle sorte qu'elle contienne un champ nommé a et qu'une méthode m de cette classe le référence. Si une classe B héritant de A ne contient pas ce champ (ce qui est possible en respectant notre formalisme) et s'il est permis d'appeler m sur une instance de B (ce qui semble possible puisque B hérite de A), alors le modèle formé des classes A et B n'est pas cohérent. De même, soit une classe A' contenant deux méthodes m_1 et m_2 telles que m_1 référence m_2. Si une classe B', sous-classe de A' hérite de m_1 et non de m_2 (ce qui est possible en respectant notre formalisme), alors l'envoi d'un message sur une instance de B' ayant pour conséquence l'exécution de m_1 peut conduire à une erreur ce qui n'est pas cohérent.

Il est possible de traiter indépendamment le problème associé au type de celui associé au comportement mais cela requiert de spécifier très précisément le concept de type, par exemple en se restreignant au type n-uplet. Dans un cadre que nous souhaitons général, il nous semble plus facile de traiter ces deux problèmes conjointement en imposant que l'accès aux données d'un objet soit effectué par le biais de méthodes spécialisées appelées accesseurs.

Pour ce faire, soit $o_{Accessor} \in O$ tel que $o_{Accessor} \prec o_{Method}$ et $o_{Accessor} \dashv o_{MetaMethod}$. Nous appellerons accesseur tout objet référencé par $o \in O$ tel que et $o \in \widetilde{\pi}(o_{Accessor})$. Notons qu'un accesseur est associé au type de la classe dans laquelle il est défini. Pour ne pas déplacer le problème de compatibilité au niveau des accesseurs, il est donc nécessaire d'imposer que les accesseurs ne puissent être hérités. De manière plus formelle, cela peut être décrit de la manière suivante :

$$\forall o \in \mathcal{M}_o, \forall m \in \mu_h(o) : m \notin \widetilde{\pi}(o_{Accessor}) \tag{2.25}$$

Pour garantir la compatibilité de classes dans le cadre de notre formalisme, il est alors nécessaire que lorsqu'une classe B hérite une méthode d'une autre classe A, alors tous les messages pouvant être envoyés aux instances de A puissent aussi être envoyés aux instances B. De manière plus formelle, cela se traduit de la manière suivante :

$$\forall o_1, o_2 \in \mathcal{M}_o, \mu_h(o_1) \cap \mu_i(o_2) \neq \emptyset \Rightarrow \rho(o_2) \subseteq \rho(o_1) \tag{2.26}$$

Enfin, pour respecter la sémantique traditionnelle du concept d'héritage, nous imposons de plus que

$$\forall o_1, o_2 \in \mathcal{M}_o, o_1 \prec o_2 \Rightarrow \rho(o_1) \supseteq \rho(o_2) \tag{2.27}$$

2.3.2.10 Compatibilité de métaclasses

Dans un langage réflexif, l'ensemble des objets peut être décomposé en plusieurs strates, appelées niveaux d'abstraction. Chaque niveau décrit et contrôle le niveau inférieur. Par exemple, les instances terminales sont contrôlées par le niveau regroupant les classes. Celles-ci sont contrôlées par le niveau composé de leurs métaclasses, elles-mêmes controlées par un autre niveau de métaclasses et ainsi de suite. L'étude de la compatibilité de métaclasses consiste à verifier que l'envoi de message entre objets de niveaux différents n'induise pas d'incohérence. Cette étude est très importante pour notre système car elle doit nous permettre de connaître les possibilités de combinaison notamment entre les classes, les vues et les relais.

Il est possible de distinguer deux types de compatibilité de métaclasses : ascendante et descendante. Dans cette partie, nous étudierons ces deux aspects et nous verrons comment formaliser les propriétés permettant de garantir la compatibilité de métaclasses. Nous baserons notre étude sur l'article [Bouraquadi et al.96].

Compatibilité ascendante Ce type de compatibilité est le plus connu. Il a par exemple été étudié dans [Graube89]. Les problèmes liés à ce type de compatibilité surviennent lorsqu'un objet envoie un message à sa classe. Prenons un exemple en nous plaçant dans un contexte d'héritage simple classique, où lorsqu'une classe A hérite d'une classe B, alors tout message pouvant être envoyé sur les objets instances de B peuvent aussi être envoyés sur les objets instances de A et conduisent à l'exécution de la même méthode. Considérons deux métaobjets o_1 et o_2 tels que o_1 puisse recevoir le message a et tels que o_2 hérite de o_1. Soient mo_1 et mo_2 les classes respectives de o_1 et o_2, Supposons que mo_1 puisse recevoir le message b et que l'exécution de la méthode provoquée par l'envoi du message a sur o_1 corresponde à envoyer le message b sur la classe du receveur du message. Si l'on envoie le message a sur o_2, il faut donc que la classe de o_2, soit mo_2 puisse recevoir le message b pour que le modèle formé des objets o_1, o_2, mo_1 et mo_2 soit cohérent. Dans le cadre d'un héritage simple classique, la vérification de la compatibilité ascendante peut-être garantie par la vérification de la propriété suivante :

- Pour toute classe A et B, si B hérite de A, alors la classe de B doit hériter de, ou être égale à la classe de A.

D'une manière plus formelle et en nous replacant dans le contexte de notre modèle, cela se traduit de la manière suivante :

- Pour tout $o_1, o_2 \in \mathcal{M}_o$ tels que $\mu_i(o_1) \cap \mu_h(o_2) \neq \emptyset$, si $\rho(\gamma(o_1)) \subseteq \rho(\gamma(o_2))$ alors la compatibilité ascendante est vérifiée.

Notons que cette propriété est une propriété suffisante mais non nécessaire. En effet, il est possible de n'avoir aucune incohérence si elle n'est pas vérifiée. De plus, cette propriété est définie en fonction du comportement intrinsèque de o_1 et du comportement hérité de o_2, et non en fonction des comportements de ces deux métaobjets ce qui aurait été plus général. Nous avons choisi cette approche car nous supposons que lorsqu'une méthode fait partie du comportement intrinsèque d'un objet, elle a été définie spécifiquement pour cet objet et il est donc de la responsabilité du programmeur de vérifier qu'elle n'induise pas d'incohérence. Par contre, lorsqu'une méthode est héritée, il est possible que le programmeur n'ai pas connaissance de son contenu, il est alors du rôle du langage de vérifier qu'elle n'induise pas d'incohérence.

Compatibilité descendante Ce type de compatibilité correspond au problème symétrique de celui décrit précédemment. Prenons un exemple en nous plaçant encore une fois dans un contexte d'héritage simple classique. Soient mo_1 et mo_2 deux métaobjets tels que mo_1 puisse recevoir le message a et que mo_2 hérite de mo_1. Soient o_1 et o_2 des instances respectives de mo_1 et mo_2. Supposons que o_1 puisse recevoir le message b et que l'exécution de la méthode provoquée par l'envoi du message a sur mo_1 corresponde à envoyer le message b sur l'ensemble des instances du receveur du message. Si l'on envoie le message a sur mo_2, il faut donc que o_2 puisse recevoir le message b pour que le modèle formé des objets o_1,o_2,mo_1 et mo_2 soit cohérent.

Dans le cadre d'un héritage simple classique, il peut être difficile de trouver une propriété simple permettant de vérifier la compatibilité descendante. Par exemple, si l'on prend l'analogie avec la compatibilité ascendante, pour toutes métaclasses A et B, si B hérite de A, alors tout classe instance de B devrait hériter de toutes les classes instances de A, ce qui est très contraignant et par ailleurs impossible dans le cadre d'un héritage simple.

De même, si l'on cherche à formaliser une propriété suffisante garantissant la compatibilité descendante, on se trouve confronté à des contraintes très fortes. Par exemple, si l'on adopte une approche similaire à celle décrite pour la compatibilité ascendante on arrive à la propriété suivante :

Pour tout $mo_1, mo_2 \in \mathcal{M}_c$ tels que $\mu_i(mo_1) \cap \mu_h(mo_2) \neq \emptyset$, si $\forall o_2 \in \pi(mo_2), \bigcup_{o_1 \in \pi(mo_1)} \rho(o_1) \subseteq \rho(o_2)$ alors la compatibilité ascendante est vérifiée.

Il semble ainsi difficile de trouver une propriété simple permettant de garantir la compatibilité ascendante dans le cadre d'un modèle réflexif à objets aussi général que celui que nous avons décrit jusqu'à présent. Par contre, il est possible de contraindre globalement le modèle pour permettre la vérification de ce type de compatibilité. C'est l'objet de la prochaine sous-partie.

2.3.2.11 Restriction du modèle

Comme nous venons de le voir, il peut être intéressant de proposer une restriction à notre formalisation permettant de définir des propriétés simples à vérifier pour garantir la compatibilité de métaclasses (ascendante ou descendante). Dans le cadre du modèle général, vérifier la compatibilité descendante nécessite de poser des contraintes fortes sur l'ensemble des objets du langage. En effet, pour garantir la compatibilité descendante, il faut prendre en compte, pour chaque métaclasse, l'ensemble des méthodes définies dans toutes les instances de cette métaclasse, celles-ci pouvant être éventuellement mises en œuvre dans l'une des méthodes de la métaclasse et induire des incohérences au niveau des instances des sous-classes de celle-ci. Ce problème est similaire dans le cadre de la compatibilité ascendante mais les contraintes à poser sont simples puisqu'il suffit pour chaque classe ou métaclasse de ne prendre en compte que les méthodes définies dans la classe de celui-ci.

Pour résoudre ce problème, deux solutions sont possibles.

- Le première est de faire en sorte que si une méthode d'une métaclasse contient un appel vers une méthode d'une de ses instances, cette métaclasse n'aura aucune sous-classe. Pris en sens inverse, cette approche revient à disposer d'une métaclasse indépendante (au sens de l'héritage) pour chaque méthode pouvant éventuellement générer un tel conflit. En pratique, lorsqu'une méthode d'une métaclasse m, se trouve dans ce cas, on crée une nouvelle

métaclasse m' héritant de m et ne pouvant pas être spécialisée par héritage. La méthode incriminée est alors déplacée au niveau de m' et toute les classes concernées deviennent des instances de m'. Pour un descriptif plus complet de cette approche, le lecteur pourra se référer à [Bouraquadi et al.96].

- La seconde consiste à limiter l'ensemble des envois de messages descendants possibles à partir d'une méthode définie au sein d'une métaclasse. Dans ce cas, il faut choisir et identifier cet ensemble des envois de messages possibles. Une solution élégante consiste à associer à chaque métaclasse une racine d'héritage pour ses instances et imposer que les méthodes définies au sein de chaque métaclasse ne puissent référencer que les méthodes définies au sein de la racine d'héritage qui lui est associée. Dans ce cas, garantir la compatibilité descendante devient aussi simple que garantir la compatibilité ascendante. Nous avons choisi de mettre en œuvre cette approche car elle nous semble plus naturelle et plus « orientée objet » que la première si l'on considère le processus de développement d'une application. Il est en effet très probable que l'ensemble des instances d'une métaclasse ne soit pas connu lors de la définition de celle-ci et donc de ses méthodes. Restreindre les envois de message descendants aux seules méthodes définies sur la racine d'héritage associée à chaque métaclasse est ainsi un bon compromis entre la généralité du modèle et les contraintes à vérifier pour garantir la compatibilité descendante. Notons de plus que cette approche s'intègre élégamment dans notre formalisme et dans la plupart des modèles à objets réflexifs précédemment décrits, en particulier si l'on considère o_{Object}, celui-ci étant par définition la racine d'héritage des instances de o_{Class}.

Pour terminer cette partie et ce chapitre, formalisons cette seconde solution. Pour ce faire, considérons la fonction, nommée $root$, de $\mathcal{M}_c$ dans $\mathcal{M}_o$ associant à chaque métaclasse la racine d'héritage de ses instances telle que $\forall o \in \mathcal{M}_c, \forall o' \in \pi(o) : o' \prec root(o)$, $root(o_{Class}) = o_{object}$, $root(o_{MetaMethod}) = o_{Method}$ et $root(o_{MetaType}) = o_{Type}$. Supposons que, pour toute métaclasse m, les méthodes définies au sein de celles-ci ne puissent référencer aucune des méthodes définies sur les instances de m sauf celles définies sur $root(m)$.

Dans ce cas, Pour tout $mo_1, mo_2 \in \mathcal{M}_c$ tels que $\mu_i(mo_1) \cap \mu_h(mo_2) \neq \emptyset$, si $\rho(root(mo_1)) \subseteq \rho(root(mo_2))$ alors la compatibilité ascendante est vérifiée.

Deuxième partie

JavaViews : Un système de programmation réflexif, paramétrable, persistant et transactionnel. Une solution pour l'intégration de sources de données hétérogènes.

Chapitre 3

Modèles et Formalisations

Dans ce chapitre, nous détaillons et nous formalisons l'intégration des différents concepts rappelés ci-après au sein d'un langage réflexif à objets en nous basant sur le modèle formel développé précédemment. Nous commencerons par le concept de transaction puis nous étendrons le formalisme obtenu aux relais, aux vues et aux requêtes.

Notons que les fonctions du modèle formel présentées dans le chapitre précédent peuvent aussi être décrites sous la forme d'envois de message puisqu'elles ne portent que sur des identifiants d'objets, des ensembles ou des listes d'identifiants d'objets, eux mêmes étant inclus dans le modèle. En conséquence, pour décrire les concepts présentés dans ce chapitre, nous pouvons au choix utiliser des fonctions mathématiques ou des messages. Nous choisirons les premières dans le cas général et les seconds lorsqu'il s'agira de définir des algorithmes complexes.

3.1 Problématique

Dans le chapitre précédent, nous avons introduit les notions d'hétérogénéité d'accès et d'hétérogénéité de stockage. Notre but est de résoudre simultanément ces deux types d'hétérogénéité en intégrant dans un langage de programmation à objets l'architecture et les techniques issues des SGMB, et ceci en adoptant une approche par couplage faible. Comme nous l'avons vu précédemment, cette intégration se traduit de manière plus précise par l'ajout au langage des concepts de transaction, de relai, de vue et de requête en respectant les propriétés suivantes :

Indépendances de type, d'accès et de comportement. Ces trois types d'indépendance doivent être respectés pour garantir l'adéquation de nos modèles à toutes les sortes de données et d'applications susceptibles d'utiliser ou de prendre part à notre système.

Factorisation. Ce paradigme garantit la plus grande souplesse d'adaptation de notre système aux besoins de ses utilisateurs. Il se traduit d'une part au niveau du concept de transaction dont les différentes composantes et propriétés sont déléguées et distribués entre des objets dédiés, les transactions, des objets applicatifs dits objets transactionnels et des propriétés, dites comportements. D'autre part, le concept de vue est introduit dans le langage sous la forme de deux types d'entités distinctes pouvant être combinées : les classes transactionnelles,

soit la structure des vues, et les mappings, soit les correspondances entre vues et classes de base.

Sémantique implicite. Pour rendre aisée l'utilisation des concepts précédemments cités, nous souhaitons aussi introduire dans le langage le concept d'envoi de message transactionnel permettant de mettre en œuvre des transactions en suivant une sémantique propre à l'objet. Pour les mêmes raisons, la sémantique du langage de requêtes doit être la plus proche possible de celle du langage de programmation acceuillant nos extensions.

3.2 Transactions

Pour l'ensemble de cette partie, nous ne considérons que les données volatiles, c'est-à-dire, nous ne prenons pas en compte les problèmes liés à la persistance des données. Nous verrons plus loin comment intégrer aux modèles développés dans cette partie une gestion des données persistantes en utilisant les propriétés d'ouverture du système.

3.2.1 Le système transactionnel

3.2.1.1 Description et Gestion des objets transactionnels

Notre approche consiste à séparer les différentes composantes transactionnelles et à définir un système transactionnel modulaire pouvant être spécialisé. L'utilisateur de notre système doit pouvoir spécifier le comportement transactionnel adéquat pour chacun des objets manipulés dans son application.

De plus il est souhaitable de laisser la possibilité de limiter l'action du système transactionnel à une partie des objets. En effet, la gestion transactionnelle des objets implique un surcoût en terme de vitesse d'exécution de l'application et d'utilisation de la mémoire par rapport à une gestion classique des objets. L'utilisateur doit être en mesure de préciser la nature de la gestion de ses objets pour garantir une efficacité maximale de son application. Nous avons choisi de permettre de spécifier la nature de la gestion des objets lors de la déclaration de leur classe. Nous décrivons ci-dessous les principaux avantages et inconvénients de cette approche.

- Du point de vue des concepts classiques de programmation par objets, tous les objets d'une même classe ont un comportement commun défini par l'ensemble des méthodes décrites dans cette classe. L'ajout de propriétés transactionnelles à un objet peut être vue comme l'ajout d'un certain comportement sur cet objet et donc par conséquence, se traduit par l'ajout du même comportement sur tous les objets instances d'une même classe[1].
- En vertu des principes d'indépendance définis précédemment et de notre volonté de proposer des sémantiques transactionnelles implicites, la gestion des transactions au sein de notre système doit être la plus transparente possible pour l'utilisateur. La définition du comportement transactionnel des objets uniquement lors de la déclaration de leur classe permet de ne plus s'occuper de gestion transactionnelle dans la logique de l'application. En conséquence, la modification du mode de gestion des objets ne nécessite pas de changer le code même de

[1] comme c'est le cas, par exemple, des contraintes en SQL3 [ISO95]

l'application, c'est-à-dire le corps des méthodes, mais de simplement modifier la déclaration des classes concernées.
- Cette approche permet aussi d'utiliser aisément une bibliothèque ayant recours à notre système transactionnel lors du développement d'applications ne faisant pas appel à notre outil tout en conservant les propriétés transactionnelles de la bibliothèque. Les principes de modularité et d'encapsulation sont ainsi respectés et la portabilité du code généré par notre outil est optimale.
- Cependant, notre approche implique de connaître, lors de la déclaration des classes, le type de gestion transactionnelle à appliquer sur les instances de celles-ci. Notamment, il est impossible de redéfinir le comportement transactionnel des objets lors de l'exécution de l'application ou dans le cadre de la réutilisation d'une bibliothèque développée avec notre outil. Nous supposons en fait que le comportement transactionnel à appliquer sur un objet dépend de sa structure et de sa sémantique interne et non de la logique générale de l'application dans laquelle il est mis en œuvre.

Nous avons vu dans le chapitre précédent que le concept de protocole à métaobjets est bien adapté à la modélisation d'une telle gestion transactionnelle lorsque chaque propriété ou composante transactionnelle est modélisée par une métaclasse. En effet, il est alors possible d'adapter le modèle transactionnel aux besoins de l'application en combinant diverses métaclasses ou en les spécialisant par héritage. Le modèle de l'application, c'est-à-dire, l'ensemble des classes de l'application, est par la suite créé en instanciant les métaclasses ad-hoc.

Pour intégrer un système transactionnel dans notre système nous définissons trois types de métaclasses :

Les métaclasses de contrôle transactionnel. Ces métaclasses sont à mettre en correspondance avec la composante de contrôle de l'exécution transactionnelle. Elles permettent, dans notre modèle, de spécifier les méthodes pouvant faire l'objet d'un envoi de message transactionnel. Dans la suite, nous appellerons de telles méthodes, méthodes transactionnelles.

Les métaclasses de gestion des objets transactionnels. Ces métaclasses sont à mettre en correspondance avec la composante de gestion des données des transactions. Nous appellerons objets transactionnels les objets, méta-instances de ces métaclasses. Notons que, comme nous l'avons précédemment précisé, les objets transactionnels n'ont pas de comportements transactionnels propres. Il leur en est associé un dynamiquement, celui-ci étant déterminé par le biais d'autres métaclasses dites de comportement définies ci-après.

Les métaclasses de comportement transactionnel. Ces métaclasses permettent de préciser le type de comportement associé à chaque objet transactionnel. Lors de la déclaration d'un classe instance d'une métaclasse de gestion des objets transactionnels, il est précisé une instance d'une métaclasse de comportement. Une instance de celle-ci, dite comportement transactionnel, est alors associée par défaut à chaque instance de la classe. Les objets transactionnels ainsi définis sont gérés lors d'une exécution transactionnelle en suivant les propriétés de son comportement transactionnel. L'utilisateur peut, s'il le souhaite, modifier cette association dynamiquement pour chaque objet transactionnel.

3.2.1.2 Envoi de message transactionnel

Comme nous l'avons vu dans la partie 2.2.4, notre système transactionnel repose aussi sur le concept d'envoi de message transactionnel. Celui-ci est constitué d'un envoi de message classique encapsulé dans une transaction (voir la figure 3.1). De plus, comme cela a été présenté en 2.2.2, une transaction peut être modélisée par un objet, instance d'une classe, que nous nommerons `Transaction`. Nous supposerons que cette classe comporte au minimum les méthodes `abort` et `commit`, de gestion de la structure d'une transaction qui constituent et définissent l'ossature du modèle transactionnel. Pour respecter notre choix d'une sémantique transactionnelle implicite, ces méthodes ne sont pas manipulées explicitement par l'utilisateur de notre système mais sont mises en œuvre automatiquement par le biais de l'envoi de message transactionnel de la façon suivante.

Si, lors de l'exécution de l'envoi de message classique une exception particulière que nous appellerons exception d'annulation est lancée, la transaction est annulée, c'est-à-dire que le message `abort` est envoyé sur l'objet transaction correspondant. Si l'exécution de l'envoi de message se termine normalement, la transaction est validée et le message `commit` est envoyé à l'objet transaction. Tous les objets transactionnels accédés entre le début de l'envoi de message transactionnel et la validation ou l'annulation de la transaction sont pris en compte par la transaction. Leur gestion au cours de la transaction ou lors de l'annulation ou la validation de la transaction, est précisée au sein du comportement transactionnel qui leur est associé. Notons ici que comme ce comportement peut-être dynamiquement modifié, nous imposons qu'il ne le soit qu'en dehors d'une transaction pour éviter tous problèmes de cohérence globale du système.

3.2.1.3 Threads

Un thread est un processus qui s'exécute dans le cadre d'une l'application et qui partage les données de l'application. Plusieurs threads peuvent s'exécuter en même temps au sein d'une même application. Il ne sont exécutés réellement en parallèle que sur des architectures multi-processeurs. Sur des machines uni-processeur, le système d'exploitation simule leur exécution en parallèle. Comme tous les threads d'une même application partagent les mêmes données, le programmeur est en règle générale obligé de gérer explicitement la concurrence sur ces données à l'aide de concepts tels que les sémaphores ou les sections critiques. Comme nous l'avons précédemment précisé, il nous semble intéressant de permettre de gérer la cohérence des données impliquées dans plusieurs threads de manière implicite, chacun d'entre eux étant alors considéré comme une transaction. Inversement, il semble aussi intéressant de permettre d'exécuter plusieurs threads au sein d'une même transaction. Au sein de notre système, cela se traduit d'une part au niveau de l'envoi de message transactionnel par la possibilité d'envoyer plusieurs messages transactionnels en parallèle (voir 3.2.d), et d'autre part, au niveau des méthodes pouvant faire l'objet d'un envoi de message transactionnel. Il doit en effet être possible que celles-ci puissent induire le démarrage de plusieurs threads (voir 3.2.e). Notons que ceci est nécessaire pour respecter l'indépendance de comportement.

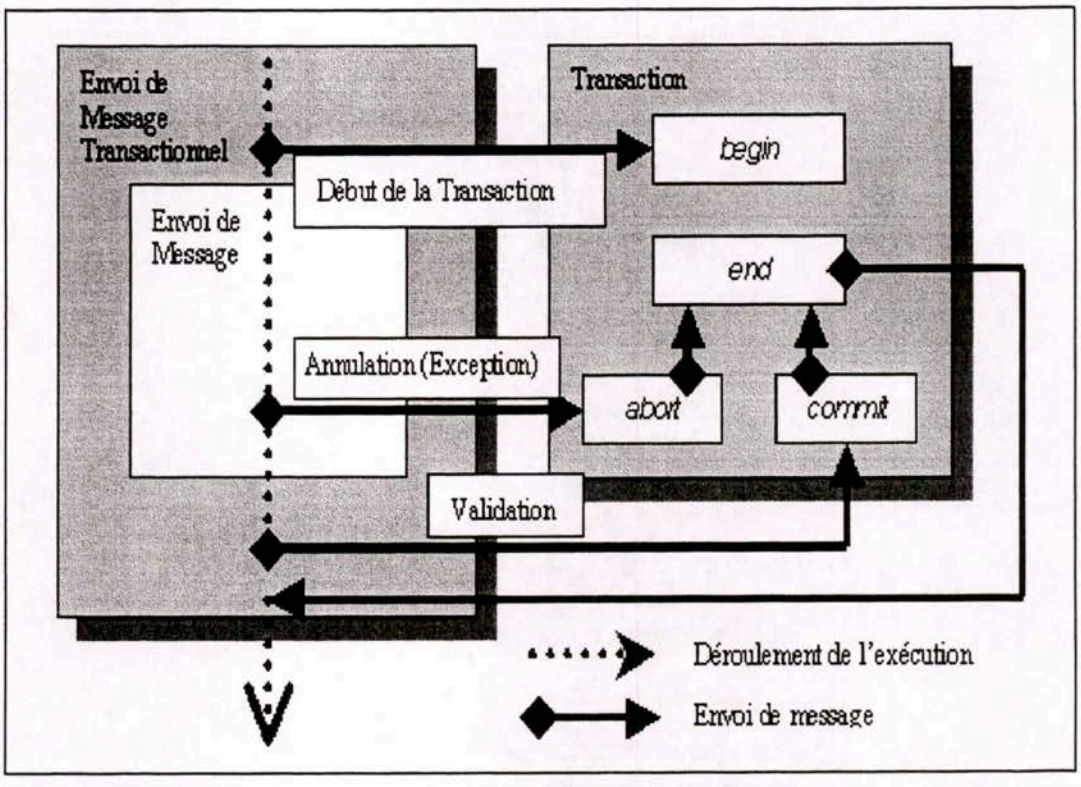


FIG. 3.1 – Envoi de message transactionnel

3.2.1.4 Imbrication

A l'instar de l'envoi de message traditionnel, il doit être possible d'imbriquer des envois de messages transactionnels. Notre modèle le permet suivant deux sémantiques distinctes, ou suivant une combinaison de ces deux sémantiques (voir figure 3.2.b et 3.2.c) :

- **Transactions imbriquées.** Comme nous l'avons déjà précisé en 1.3.1.3, ce modèle consiste à enrichir le modèle transactionnel plat classique en permettant que chaque transaction contienne des sous-transactions, qui, à leur tour peuvent être composées de sous-transactions. L'arbre transactionnel ainsi formé doit respecter les règles suivantes : (a) l'annulation d'une transaction implique l'annulation de ses sous-transactions. (b) La validation d'une sous-transaction n'est visible que dans l'environnement de sa transaction mère. (c) Une transaction ne peut être validée tant que toutes ses sous-transactions ne sont pas terminées.
- **Transactions de haut niveau.** Le modèle précédent présente une disymétrie. L'annulation d'une transaction implique l'annulation de ses sous-transactions, l'inverse n'étant pas vrai. Pour une plus grande souplesse et une plus grande généralité de notre système, nous avons souhaité rendre les sous-transactions indépendantes de la validation ou de l'annulation de leur transaction mère. Ce modèle est identique au modèle précédent à l'exception du mode de terminaison des transactions. La règle (a) est supprimée. La règle (c) est étendue : Une transaction ne peut etre validée ou annulée tant que toutes ses sous-transactions ne sont pas terminées. La contrainte sur l'annulation n'est pas nécessaire dans le modèle précédent à cause de la règle (a). Notons que la contrepartie à la plus grande souplesse de ce nouveau modèle transactionnel est une perte de la sémantique forte qu'il peut exister entre

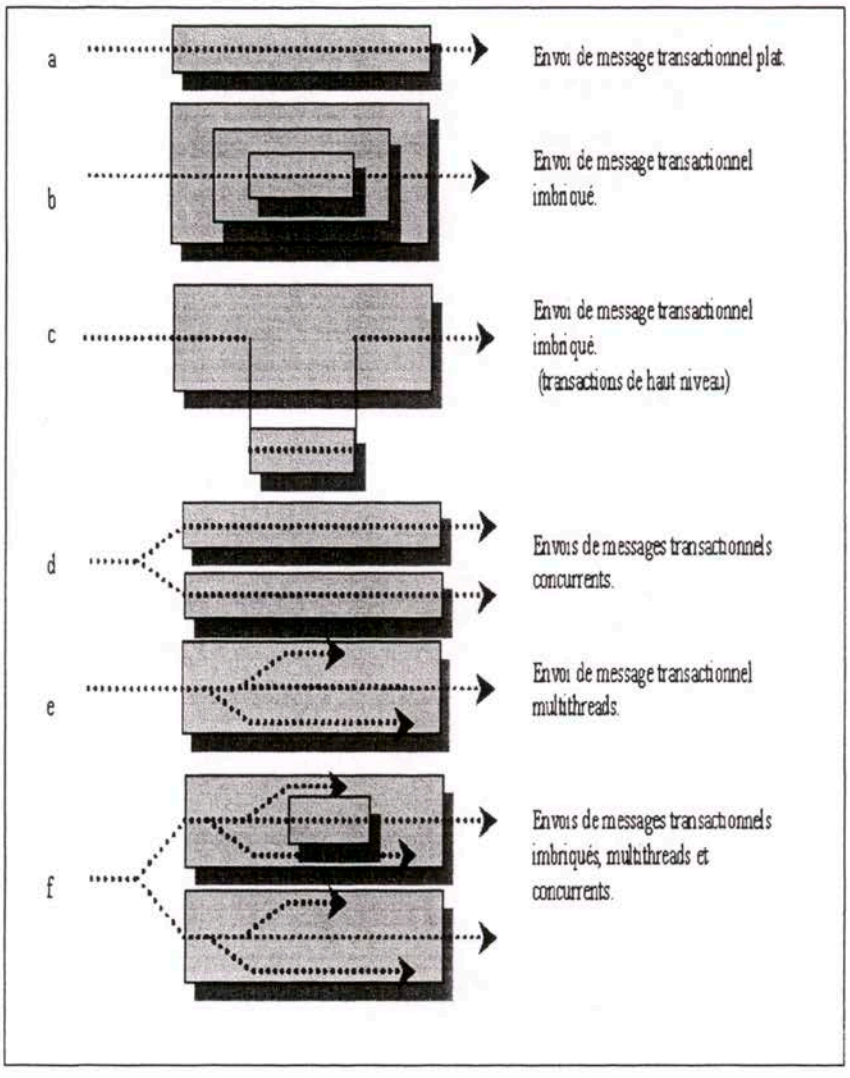


FIG. 3.2 – Types d'envois de messages Transactionnels

transaction mère et transaction fille dans le modèle des transactions imbriquées.

Enfin, notre modèle permet de combiner gestion des threads et envoi de messages transactionnels imbriqués (voir figure 3.2.f).

3.2.2 Modèle transactionnel

Dans cette partie, nous décrivons plus précisément le modèle transactionnel que nous avons choisi ainsi que les différentes interactions entre les entités prenant part à ce modèle. Pour ce faire, nous utilisons une extension du formalisme graphique de description de modèles transactionnels présentés en [Gray et al.93].

3.2.2.1 Formalisme graphique

La figure 3.3 présente notre formalisme graphique. Une unité transactionnelle y est réprésentée par une boîte. Le tiers supérieur de cette boîte décrit l'interface publique de la transaction. Cette interface décrit, sous forme d'hexagones, les actions pouvant être invoquées par d'autres entités, transactionnelles ou non, extérieures à la transaction. Elle peut inclure par exemple les actions `commencer` (B), `annuler` (A) ou `valider` (C). Le tiers inférieur de la boîte est constitué des états terminaux possibles de l'entité transactionnelle considérée. Pour une transaction classique, cela peut être `validée`, `annulée` et `terminée`. Ces états sont représentés par des pentagones. La partie médiane de la boîte contient, sous forme d'ellipses, les différents comportements internes, c'est-à-dire les méthodes utilisées dans le processus de validation et d'annulation de l'entité transactionnelle. Enfin, le carré central représente le processus interne de l'entité transactionnelle en considérant celle-ci comme un agent autonome pouvant agir sur les différents éléments présentés précédemment. Ce carré peut aussi être vu comme représentant les effets du programme utilisateur restreints à l'entité transactionnelle considérée. Alors que le formalisme présenté en [Gray et al.93] permet de décrire les liens sémantiques entre les différentes entités transactionnelles mises en jeu dans le cadre de modèles transactionnels traditionnels, notre symbolisme permet de plus de décrire des sémantiques transactionnelles très complexes ainsi que le comportement interne des entités transactionnelles.

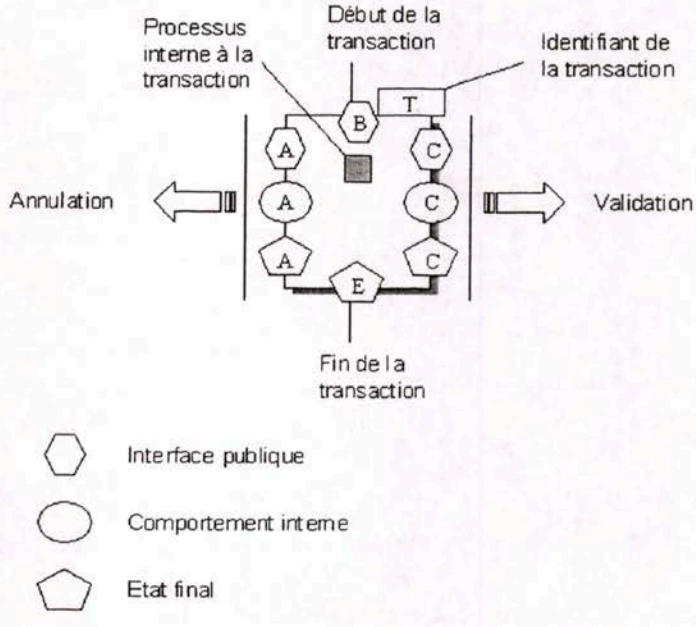


FIG. 3.3 – Formalisme graphique

3.2.2.2 Transactions imbriquées

La figure 3.4 représente le modèle transactionnel décrit précédemment en ne considérant que les entités transactionnelles suivantes :

- les transactions de haut-niveau (T) ;

- les sous transactions imbriquées (STI) ;
- les sous-transactions de haut-niveau ($STHA$).

Une flèche en pointillés représente la nécessité d'attendre la fin de l'action ou l'état correspondant à l'origine de la flèche pour exécuter l'action située à l'extrémité de la flèche. Une flèche pleine indique une implication logique, c'est-à-dire que l'exécution de l'action, origine de la flèche, implique l'exécution de l'action décrite à l'extrémité de la flèche. Notons que ceci peut-être implanté sous la forme d'un envoi de message dans le cadre de la programmation par objets. Les flèches pleines peuvent être extérieures aux boîtes et expriment alors une relation entre plusieurs entités transactionnelles. Lorqu'elles sont intérieures, elles décrivent le comportement interne de ces entités.

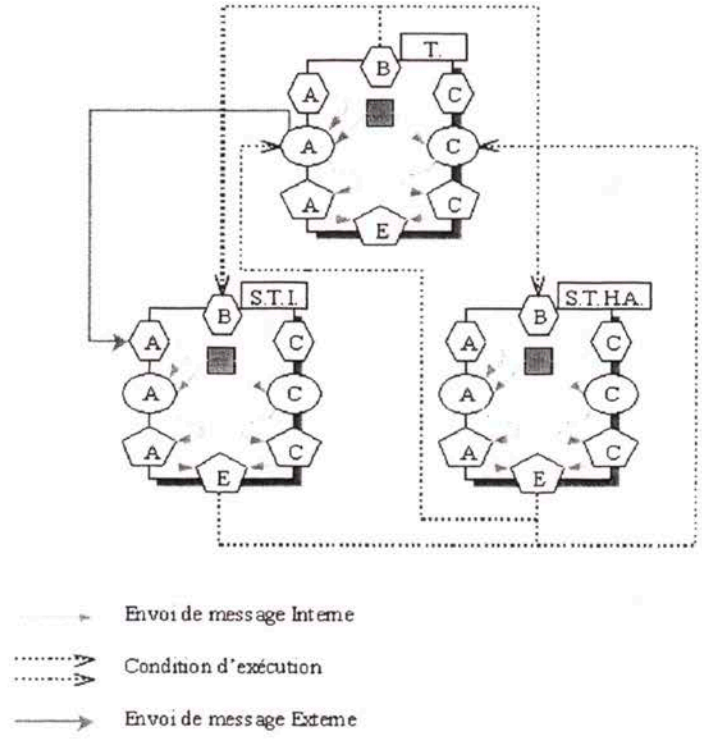


FIG. 3.4 – Envois de messages transactionnels imbriqués

Détaillons cette figure. Tout d'abord, il est évident, ce qui est représenté par les flèches en pointillés reliant les hexagones B, que les sous-transactions ne peuvent commencer tant que leur transaction mère n'a pas commencé. De plus, pour respecter le modèle des transactions imbriquées tel que présenté précédemment, il est nécessaire que l'annulation d'une transaction induise l'annulation de ses sous-transactions imbriquées. Cela est représenté par la flèche pleine reliant l'ellipse A de la transaction T à l'hexagone A de la sous-transaction STI. Pour en terminer avec le comportement externe des entités transactionnelles représentées ici, il est à noter que, toujours pour respecter le modèle des transactions imbriquées, une transaction ne peut être validée tant que toutes ses sous-transactions ne sont pas terminées et ne peut être annulée tant que toutes ses sous-transactions de haut niveau ne sont pas teminées. Ceci est représenté par les flêches pointillées prenant leur origine sur les pentagones de STI et $STHA$.

Considérons à présent le comportement interne des entités transactionnelles et plus particulièrement la partie inférieure de leur représentation. Notons tout d'abord que celui-ci est identique pour toutes les entités transactionnelles représentées sur la figure 3.4. C'est d'ailleurs la raison pour laquelle le formalisme présenté en [Gray et al.93], dédié aux entités transactionnelles traditionnelles, ne permet de le représenter. Intéressons nous à la partie gauche de la représentation de ce comportement. Celle-ci nous indique que lorsqu'une transaction reçoit un message d'une autre transaction lui demandant d'annuler (hexagone A), le comportement d'annulation (ellipse A) doit alors être exécuté conduisant à placer l'état de l'entité transactionnelle à `annulée` (pentagone A) puis `terminée` (pentagone E). Notons que l'exécution du processus interne peut conduire au même résultat. Le côté droit de la représentation du comportement interne concerne la validation. Il est similaire au côté gauche excepté l'absence de flèche entre l'hexagone et l'ellipse C, ce qui indique qu'aucune entité extérieure ne peut demander la validation de l'une de ces entités transactionnelles. Pour terminer cette partie, notons que, en suivant les règles du modèle des transactions imbriquées, si l'annulation d'une STI peut être provoquée par une autre entité transactionnelle, l'annulation d'une transaction ou d'une sous-transaciton de haut niveau ne peut être déclenchée que par l'entité transactionnelle elle-même. En conséquence, il ne devrait pas y avoir de flèche entre l'hexagone et l'ellipse A de $STHA$. Nous verrons dans les parties suivantes que de nouvelles entités transactionnelles peuvent tout de même envoyer un message d'annulation à une (sous-)transaction de haut niveau. C'est pourquoi chaque boîte transactionnelle présente un lien entre l'hexagone A et l'ellipse A.

3.2.2.3 Thread

Lorsqu'un thread est démarré dans une méthode transactionnelle, il doit être considéré comme une entité transactionnelle pour permettre la gestion de la concurrence sur les données manipulées au sein du thread et donc garantir les propriétés transactionnelles que doit vérifier l'exécution de la méthode considérée. D'autre part, un thread est une unité d'exécution autonome. Une des conséquences de cette caractéristique est que les exceptions lancées au sein d'un thread ne peuvent être attrapées à l'extérieur du thread. Dans notre modèle, l'annulation d'une transaction est déclenchée par le lancement d'une exception transactionnelle. Le thread doit donc être une entité transactionnelle active pour attraper ce type d'exception et la traiter.

Dans notre modèle, nous souhaitons adopter une gestion des threads, démarrés au sein d'une transaction, similaire à la gestion des transactions imbriquées et ceci pour deux raisons. Tout d'abord, pour permettre la gestion transactionnelle des objets mis en œuvre au sein d'un thread, celui-ci doit s'exécuter dans l'environnement de la transaction depuis lequel il a été lancé, c'est-à-dire depuis sa transaction mère. De plus, toute transaction ne peut être validée tant que tous les threads démarrés dans son environnement ne sont pas terminés. En effet, dans le cas contraire, il serait impossible de déterminer l'environnement d'exécution du thread. Cependant, un thread ne peut être considéré comme une unité de reprise à l'image d'une sous-transaction du modèle des transactions imbriquées, c'est-à-dire que l'annulation des effets du thread doit impliquer l'annulation de sa transaction mère. En effet, comme une méthode transactionnelle peut-être exécutée dans le cadre d'un envoi de message transactionnel comme dans le cadre d'un envoi de message classique, le démarrage d'un thread ne doit pas porter, du point de vue de l'utilisateur, une sémantique transactionnelle particulière. En fait, les threads ne sont gérés suivant notre modèle transactionnel que dans le cadre d'un envoi de message

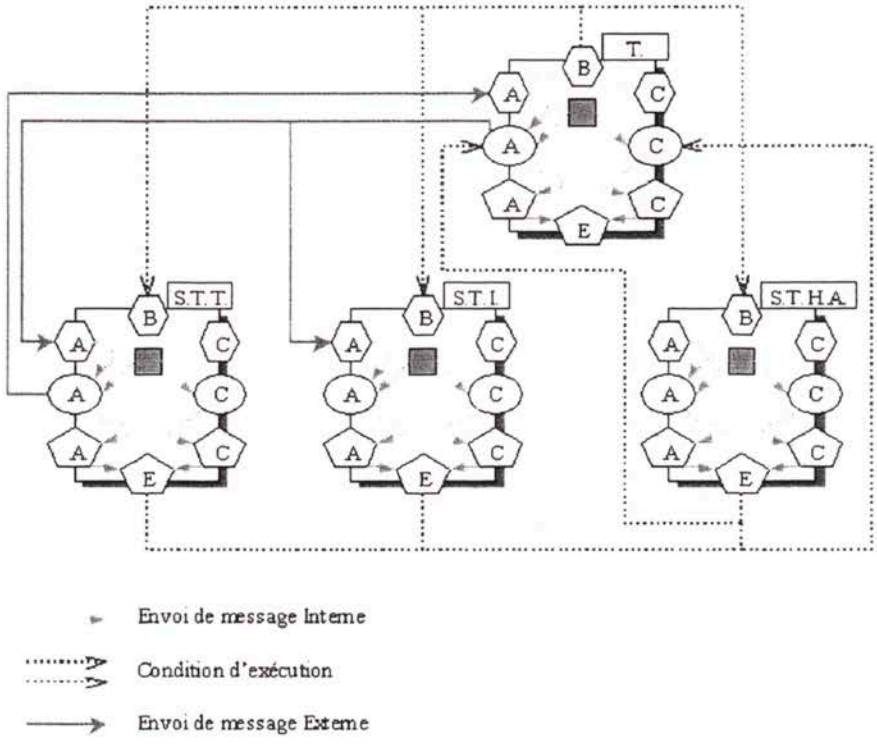


FIG. 3.5 – Threads transactionnels

transactionnel et ce, uniquement pour garantir de bonnes propriétés à notre modèle.

En conséquence, dans notre modèle, chaque thread exécuté dans un environnement transactionnel est associé à une entité transactionnelle portant une sémantique différente des deux types de sous-transactions précédemment décrits. Dans la suite nous nommerons ce type de sous-transactions : sous-transaction de thread (STT). Comme chaque sous-transaction de thread est fortement liée à sa transaction mère, à l'inverse des sous-transactions imbriquées ou de haut niveau, il n'est pas nécessaire de gérer un arbre de sous-transactions de thread. En conséquence, si un thread est démarré dans l'environnement d'une sous-transaction de thread $s1$, une nouvelle sous-transaction de thread est créée et est alors considérée comme sous-transaction de la transaction mère de s_1 et non pas de s_1. En d'autres termes, lors du démarrage d'une transaction de thread, celle-ci s'enregistre auprès de la transaction courante mais ne devient pas la transaction courante comme cela peut être le cas pour les autres types de sous-transaction.

La figure 3.5 décrit de manière graphique notre modèle transactionnel limité aux trois types de sous-transactions présentées jusqu'à présent. L'implication reliant l'ellipse A de la transaction T et l'hexagone A de la sous-transaction STT indique que, lors de l'annulation d'une transaction, d'une part les effets de ses sous-transactions de thread doivent être annulés mais qu'il est aussi nécessaire, pour éviter d'exécuter des actions qui n'auraient de toute manière aucun effet, de stopper le thread associé à la sous-transaction.

3.2.2.4 Redéfinition de l'envoi de message transactionnel

La gestion des threads au sein de notre modèle a une incidence sur la sémantique de l'envoi de message transactionnel. Considérons une méthode transactionnelle dans laquelle sont démarrés deux threads t_1 et t_2. t_1, t_2 et le thread t_m, dans lequel s'exécute la méthode transactionnelle, sont donc concurrents et s'exécutent simultanément. Supposons qu'une exception transactionnelle soit lancée au sein de t_2 alors que t_1 et t_m continuent de s'exécuter, ceci provoque l'annulation de la sous-transaction de thread associée à t_2 et par conséquent, en suivant les règles d'annulation décrites dans la figure 3.5, cela provoque l'annulation de la transaction associée à t_m et, par la suite l'annulation de celle associée à t_1. Dans tous les cas, annuler une sous-transaction ou, plus généralement, une transaction consiste en partie à en terminer l'exécution. Lorsqu'une exception est lancée, l'exécution courante, par exemple celle composée de l'ensemble des instructions d'un bloc *try* en Java, est automatiquement terminée. Cependant, l'annulation d'une sous-transaction peut impliquer l'annulation de n autres transactions. Comme il n'est pas possible de provoquer le lancement d'une exception depuis l'extérieur d'un thread, il faut donc, pour annuler ces n autres transactions, stopper l'exécution de leur thread. Revenons à notre exemple. Comme nous venons de le voir, si la transaction associée à t_1 est annulée, il faut alors stopper les threads t_2 et t_m. Si l'arrêt de t_2 ne pose pas de problème, il en est autrement pour t_m. En effet, t_m est le thread dans lequel est exécuté la méthode transactionnelle, nous voulons juste stopper l'exécution de cette méthode et non stopper l'exécution du thread t_m en entier, celui-ci pouvant contenir d'autres instructions.

Pour résoudre ce problème, deux solutions sont possibles :

- Attendre la fin de l'exécution de la méthode transactionnelle avant de continuer la phase d'annulation ;
- Faire en sorte que la méthode transactionnelle s'exécute dans un thread à part.

La deuxième solution nous semble meilleure. En effet, le temps d'exécution pour démarrer le nouveau thread nous paraît pouvoir être négligé, dans la majorité des cas, par rapport au temps d'attente de la fin de l'exécution de la méthode. De plus, cette solution est très adaptée à notre modèle puisqu'elle correspond à démarrer une nouvelle sous-transaction de thread chargée d'exécuter la méthode transactionnelle.

En conséquence, un envoi de message transactionnel se réduit à :

1. démarrer une transaction T (qui peut être considérée comme sous-transaction par une autre transaction).
2. Démarrer une sous-transaction de thread chargée d'exécuter la méthode transactionnelle.
3. Attendre la fin de la transaction T.
4. Renvoyer le résultat de la méthode transactionnelle si T a été validée.

3.2.2.5 Objets et Comportements Transactionnels

Nous avons décrit précédemment les différentes structures transactionnelles pouvant être impliquées dans l'exécution d'une application ainsi que les règles garantissant une exécution transactionnelle correcte. Dans cette partie, nous abordons la gestion du comportement transactionnel de l'application et plus particulièrement, les effets d'une exécution transactionnelle

sur les données de l'application. Nous avons vu que le comportement transactionnel de l'application, c'est-à-dire la sémantique des opérations de validation et d'annulation, était précisé lors de la description des classes, chaque objet transactionnel pouvant avoir un comportement transactionnel propre. La sémantique de l'annulation ou de la validation de chaque transaction dépend donc entièrement des objets transactionnels manipulés au cours de l'exécution de cette transaction et des comportements transactionnels qui leur sont associés. Elle ne peut donc pas être définie comme un comportement de la transaction mais doit être déléguée à chaque objet.

Par souci de simplification, nous considérons dans cette partie qu'un objet transactionnel et son comportement transactionnel associé ne forment qu'une seule entité nommée objet transactionnel. Cette simplification est sans perte de généralité dans le sens où d'une part, l'utilisateur ne gère pas directement les comportements transactionnels, et d'autre part, il est impossible de modifier le comportement transactionnel d'un objet transactionnel lorsque celui-ci est mis en œuvre au sein d'une transaction.

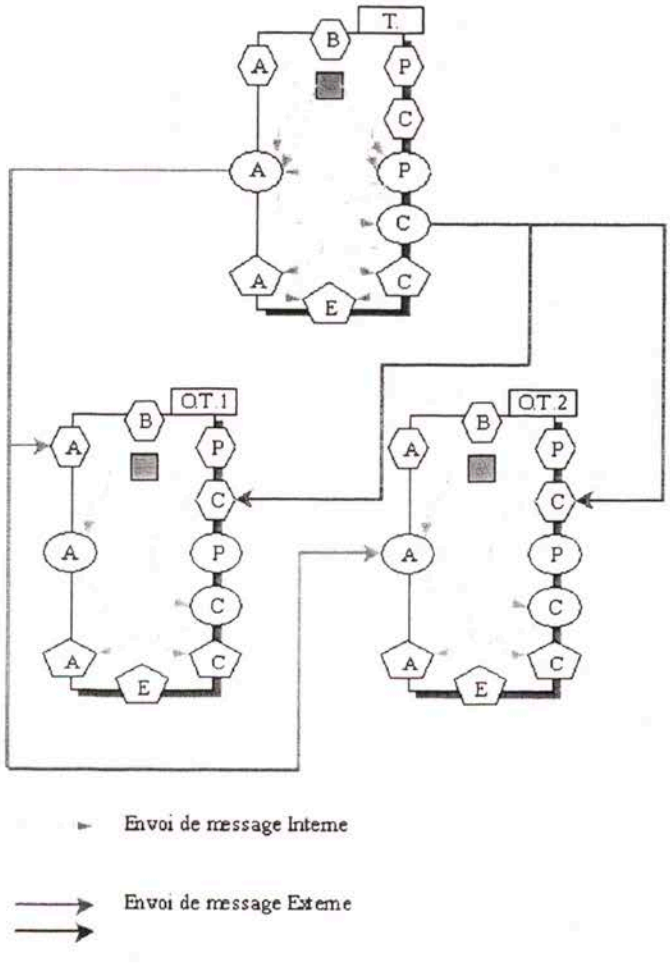


FIG. 3.6 – Objets Transactionnels

La délégation de la sémantique d'annulation ou de validation d'une transaction à plusieurs entités transactionnelles distinctes amène au problème lié à la nécessité d'atomicité de la

procédure de validation. Nous avons déjà abordé ce problème dans le cas des transactions distribués. Pour le résoudre, il est possible d'utiliser le protocole de validation en deux phases (2PC). Pour mémoire, celui-ci spécifie que lorsqu'une transaction, dite coordinatrice, délègue sa validation à plusieurs autres transactions, l'opération de validation est décomposée en deux phases :

- la phase de préparation : la transaction coordinatrice envoie une message de préparation à chaque transaction. Chacune vérifie alors si la validation est possible. Si c'est le cas, elle renvoie un message d'acquittement à la transaction coordinatrice, sinon elle renvoie un message de refus.
- la phase de terminaison : si toutes les transactions ont envoyé le message d'acquittement, elles doivent alors toutes être validées. Dans le cas contraire, elles doivent toutes être annulées. En d'autres termes, elles doivent obéir à la décision prise par la transaction coordinatrice en fonction de leur réponse. De cette manière soit toutes les transactions sont validées soit toutes les transactions sont annulées.

Dans notre modèle, nous considérons chaque objet transactionnel comme une sous-transaction à laquelle est déléguée une partie du comportement de validation ou d'annulation. Lorsqu'un objet est accédé dans l'environnement d'une transaction, il s'enregistre auprès de la transaction active. La phase de validation de la transaction se déroule ensuite suivant le modèle de validation en 2 phases. Chaque objet enregistré vérifie s'il peut être validé. Si c'est le cas, la transaction envoi le message de validation à chaque objet. Sinon, la transaction entre en phase d'annulation. Nous avons vu précédemment qu'une sous-transaction imbriquée était validée dans l'environnement de sa transaction mère. Du point de vue des objets transactionnels cela implique que chacun d'entre eux doit, après la phase de validation d'une sous-transaction imbriquée, être automatiquement enregistré auprès de la transaction mère de celle-ci. Les objets transactionnels sont donc des sous-transactions particulières, d'une part pouvant être validées et annulées plusieurs fois et, d'autre part, n'étant pas liées à une seule transaction mère.

Ces règles sont transcrites graphiquement dans la figure 3.6. Celle-ci représente deux objets transactionnels $OT1$ et $OT2$ impliqués dans une transaction T. On peut noter qu'il n'y a plus de conditions d'exécution car les objets transactionnels peuvent être impliqués dans plusieurs transactions et sous-transactions à la fois et que leur existence n'est pas liée à celle d'une entité transactionnelle particulière.

Pour prendre en compte le processus de validation à 2 phases, nous avons étendu notre formalisme en graphique en ajoutant les opérations de préparation - le losange et l'ellipse P - à chaque boîte. Le comportement interne de chaque transaction a aussi été modifié pour prendre en compte cette extension. Par exemple, on voit que le comportement interne de préparation peut conduire soit à l'annulation soit à la validation de la transaction. Notons aussi que l'implication reliant l'ellipse P de la transaction T aux hexagones P des objets transactionnels est à double sens. Cela indique que chaque objet transactionnel doit retourner une réponse à la transaction T. Cette réponse permet à la transaction T de décider si elle doit être validée ou annulée.

Un objet transactionnel ne peut décider par lui-même sa propre annulation ou validation car ces actions doivent être effectuées dans le cadre d'une transaction. En conséquence, le processus interne des sous-transactions $OT1$ et $OT2$ n'est relié à aucun comportement comme on peut le voir sur la figure 3.6. De plus, comme un objet transactionnel ne peut avoir de sous-transaction, le comportement interne de préparation n'a aucun effet sur l'objet, ce qui

est représenté par l'absence de lien ayant comme origine l'ellipse P des objets transactionnels. Enfin un objet transactionnel ne peut se retrouver dans l'état terminé car il peut être annulé et validé plusieurs fois. Il n'y a donc pas de lien entre les pentagones A et C et le pentagone E.

3.2.2.6 Résumé du modèle

Pour résumer, notre modèle transactionnel est basé sur le modèle des transactions imbriquées et l'étend en prenant en compte plusieurs types de sous-transactions.

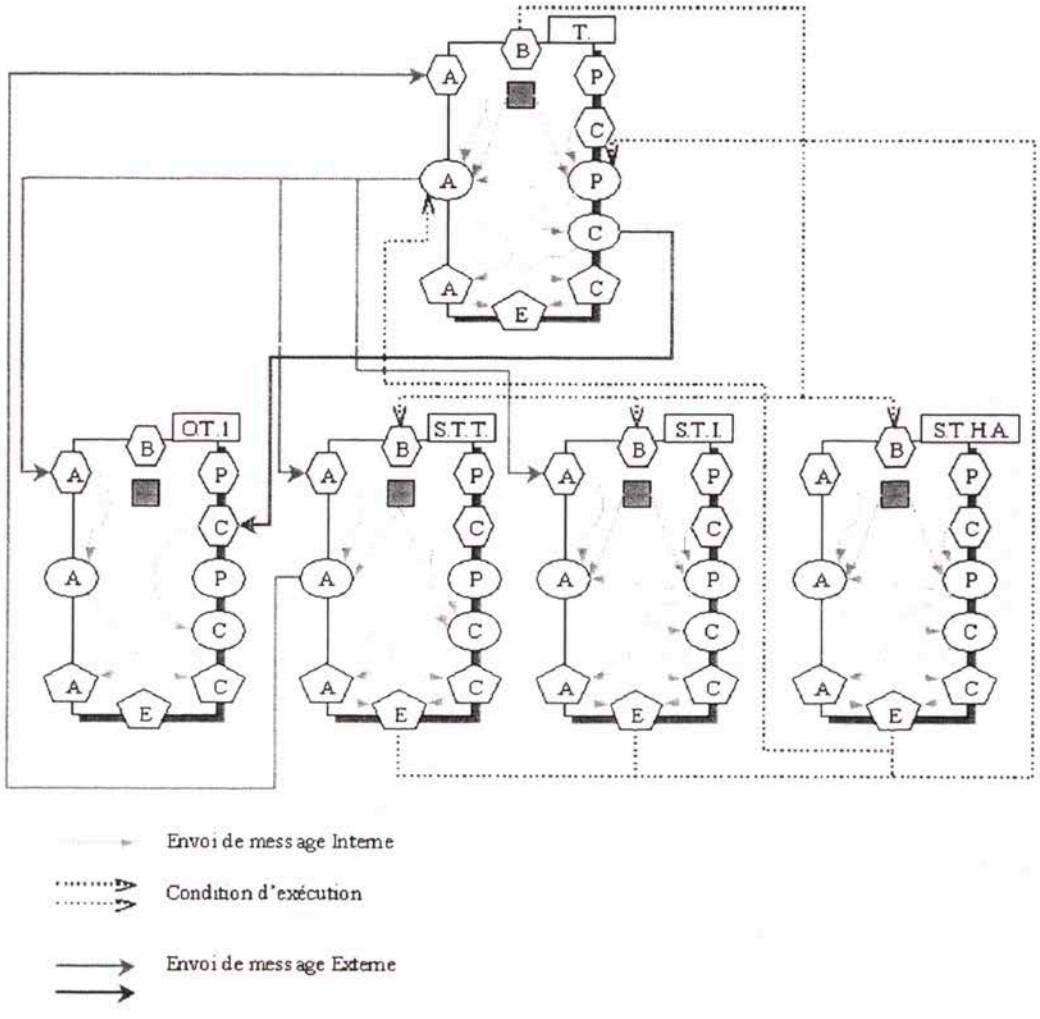


FIG. 3.7 – Résumé du modèle

- **Les sous-transactions imbriquées** (STI) : elles correspondent aux sous-transactions du modèle imbriqué classique. Elles peuvent annuler ou valider indépendamment de leur transaction mère. Néanmoins, l'abandon de leur transaction mère implique leur annulation. Enfin la transaction mère d'une STI ne peut valider sans que celle-ci soit terminée.

- **Les sous-transactions de haut niveau** ($STHA$) : elles sont une extension des sous-transactions imbriquées permettant de relâcher la propriété d'atomicité. Elles ont le même comportement que celles-ci, mis à part le fait que l'abandon de la transaction mère d'une $STHA$ d'une part ne peut être déclenché tant que celle-ci n'est pas terminée et, d'autre part, n'implique pas son annulation.
- **Les sous-transactions de thread** (STT). Pour garantir une exécution transactionnelle correcte dans un environnement multithread, nous avons défini un nouveau type de sous-transaction. Une telle sous-transaction est similaire à une sous-transaction imbriquée. Elle doit néanmoins vérifier deux contraintes supplémentaires. Son annulation implique l'annulation de sa transaction mère et elle ne peut contenir d'autres sous-transactions.
- **Les objets transactionnels** (OT). Dans le but de déléguer une partie de la validation aux objets transactionnels pour une plus grande souplesse, nous considérons chacun d'entre eux comme une sous-transaction. Nous avons déterminé que la validation d'une transaction faisant intervenir des objets transactionnels devait être réalisée suivant le protocole de validation en 2 phases. En tant que sous-transaction, les objets transactionnels ne peuvent être validés ou annulés unilatéralement et ne peuvent contenir d'autres sous-transactions.

La figure 3.7 résume l'ensemble de notre modèle.

3.2.3 Formalisme

Plaçons nous à présent dans le contexte du formalisme présenté dans le chapitre précédent et décrivons les différents concepts et modèles que nous venons d'introduire.

3.2.3.1 Envoi de message transactionnel

L'envoi de message transactionnel nécessite la mise en œuvre de plusieurs entités et techniques :

- Des métaclasses qui permettent de préciser la nature de l'envoi de message possible sur les méthodes qui y sont décrites.
- Des classes modélisant un modèle transactionnel
- Un nouveau mode d'envoi de message

Métaclasses de contrôle transactionnel Soient $o_{TransactionAbleClass} \in \mathcal{M}_c$ et $o_{TransactionAbleObject} \in \mathcal{M}_o$ tels que :

- $o_{TransactionAbleObject} = root(o_{TransactionAbleClass})$;
- $o_{TransactionAbleClass} \prec o_{Class}$ et $o_{TransactionAbleClass} \dashv o_{Class}$;
- $o_{TransactionAbleObject} \prec o_{Object}$ et $o_{TransactionAbleObject} \dashv o_{TransactionAbleClass}$.

Nous appellerons **métaclasse de contrôle transactionnel** tout objet référencé par $o \in \mathcal{M}_c$ tel que $o \prec o_{TransactionAbleClass}$. Nous appellerons **classe de contrôle transactionnel** tout objet référencé par $o' \in \mathcal{M}_o$ tel que $o' \prec o_{TransactionAbleObject}$ et $o' \in \widetilde{\pi}(o_{TransactionAbleClass})$. Soit $\mathcal{M}_{ct}$ l'ensemble des classes de contrôle transactionnel, nous appellerons **fonction de contrôle transactionnel**, notée τ, une fonction de $\mathcal{M}_{ct}$ dans $P^{fin}(\mathcal{B})$ associant à chaque classe de contrôle transactionnel l'ensemble de ses méthodes pouvant faire l'objet d'un envoi de message transactionnel, telle que $\forall o \in \mathcal{M}_{ct}, \tau(o) \subseteq \mu(o)$.

Enfin, nous appellerons O_{ct}, l'ensemble des objets de contrôle transactionnel, c'est-à-dire le sous-ensemble de O défini de la manière suivante :

$$O_{ct} = \{o \in O \mid \gamma(o) \in \mathcal{M}_{ct}\} \tag{3.1}$$

Transactions Considérons l'objet référencé par $o_{Transaction}$ tel que :

- $o_{Transaction} \prec o_{Object}$ et $o_{Transaction} \dashv o_{Class}$;
- les messages *begin*, *commit*, *abort*, *end*[2] sont éléments de $\rho(o_{Transaction})$.

Nous appellerons **transaction** tout objet référencé par $o \in O$ tel que $o \in \widetilde{\pi}(o_{Transaction})$.

Notre modèle transactionnel, construit à partir du modèle des transactions imbriquées, permet de mettre en œuvre des transactions imbriquées concurrentes. Il est aussi possible que l'utilisateur souhaite spécialiser notre système en définissant de nouveaux modèles transactionnels et donc mette en œuvre d'autres types de transactions. Plutôt que de formaliser le modèle transactionnel tel que présenté dans la partie précédente, nous préférons décrire le concept de transaction de manière générale et évolutive.

Nous baserons notre étude sur les propriétés suivantes permettant la description de nombreux modèles transactionnels dont celui présenté dans la partie précédente :

- Une transaction peut éventuellement être exécutée dans l'environnement d'une autre transaction (c'est le cas, par exemple, des sous-transactions imbriquées et des sous-transactions de haut niveau).
- Une transaction peut éventuellement être validée dans l'environnement d'une autre transaction (c'est le cas, par exemple, des sous-transactions imbriquées).
- Une transaction peut éventuellement être annulée dans l'environnement d'une autre transaction.
- Une transaction peut posséder des sous-transactions.
- Une transaction peut accéder à de nombreux objets transactionnels.
- L'exécution d'une transaction peut-être suspendue.
- L'exécution d'une transaction peut-être redémarrée.
- Une transaction peut devoir attendre la terminaison d'autres transactions pour être validée.
- Une transaction peut devoir attendre la terminaison d'autres transactions pour être annulée.

Ces différentes propriétés peuvent se traduire au niveau formel par les fonctions suivantes :

- Pour prendre en compte les trois premières propriétés, les fonctions *getBegin*, *getAbort*, et *getCommit* de $\widetilde{\pi}(o_{Transaction})$ dans $\widetilde{\pi}(o_{Transaction})$, associent respectivement à chaque transaction, la transaction dans laquelle elle commence, celle dans laquelle elle est annulée et celle dans laquelle elle est validée. Lorsque le résultat de ces fonctions est o_{Void}, cela indique que la transaction effectue l'action considérée au plus haut niveau (valeur par défaut).
- Les deux propriétés suivantes sont quant à elles relatives aux différents objets pouvant être impliqués dans une transaction. Nous définissons en conséquence les fonctions *getSubs* et *getObjects* de $\widetilde{\pi}(o_{Transaction})$ dans $P^{fin}(\widetilde{\pi}(o_{Transaction}))$ associant à chaque transaction l'ensemble de ses sous-transactions et l'ensemble des objets transactionnels auxquels elle a accédé.

[2] Lorsque nous décrirons un message nous omettrons les « » autour du nom du message.

- Les deux propriétés suivantes doivent permettre à certains algorithmes de mettre en attente puis redémarrer une transaction pour effectuer certaines tâches. Nous ajoutons en conséquence deux messages pouvant être envoyés à une transaction qui sont *suspend* et *resume*.
- Enfin pour prendre en compte les deux dernières propriétés, nous définissons les fonctions $waitForCommit$ et $waitForAbort$ de $\widetilde{\pi}(o_{Transaction})$ dans $P^{fin}(\widetilde{\pi}(o_{Transaction}))$ associant à chaque transaction l'ensemble des sous-transactions vérifiant respectivement l'une des deux propriétés correspondantes.

Enfin, définissons les fonctions $abortPriority$, $preparePriority$ et $commitPriority$, allant de $\widetilde{\pi}(o_{Transaction})$ dans $\widetilde{\pi}(o_{integer})$ associant à chaque transaction un entier. Lorsqu'une transaction est validée, annulée ou en phase de préparation, elle doit envoyer les messages correspondant à chacune de ces sous-transactions concernées. Les entiers résultants de ces fonctions permettent de trier ces sous-transactions afin de leur envoyer ces messages dans un ordre précis. Nous verrons plus loin (3.3.2.3) l'utilité de cet ordre.

Fonction d'exécution transactionnelle Nous appellerons fonction d'exécution transactionnelle une fonction, nommée λ_t, de $O_{ct} \times \mathcal{B} \times L^{fin}(O)$ dans O telle que pour tout $o \in O_{ct}$, pour tout $m \in \tau(\gamma(o))$, pour tout $a \in L^{fin}(O)$ tel que $a \ll sig(m)$, $\exists! t \in \widetilde{\pi}(o_{Transaction}) \setminus \{o_{Void}\}$, tel que[3] :

- $\lambda_t(o, m, a) \Rightarrow \lambda'(t, begin, ())$
- $\lambda_t(o, m, a) \Rightarrow \lambda(o, m, a)$
- $\lambda_t(o, m, a) = \lambda(o, m, a) \Leftrightarrow \lambda'(t, commit, ())$
- $\lambda_t(o, m, a) = o_{Void} \Leftarrow \lambda'(t, abort, ())$

Nous dirons que t est la transaction associée à l'envoi de message considéré.

Fonction d'envoi de message transactionnel Nous appellerons fonction d'envoi de message transactionnel, une fonction, notée λ'_t, de $O_{ct} \times \mathcal{S} \times L^{fin}(O)$ dans O telle que pour tout $o \in O_{ct}$, pour tout $s \in \rho(\gamma(o))$ telle que $\alpha(\gamma(o))(s) \in \tau(\gamma(o))$, pour tout $a \in L^{fin}(O)$ tel que $a \ll sig(\alpha(\gamma(o))(s))$, $\exists! t \in \widetilde{\pi}(o_{Transaction}) \setminus \{o_{Void}\}$, tel que :

- $\lambda'_t(o, s, a) = \lambda_t(o, \alpha(\gamma(o))(s), a)$
- $\lambda'_t(o, s, a) \Rightarrow \lambda'(t, begin, ())$
- $\lambda'_t(o, s, a) = \lambda(o, \alpha(\gamma(o))(s), a) \Leftrightarrow \lambda'(t, commit, ())$
- $\lambda'_t(o, s, a) = o_{Void} \Leftarrow \lambda'(t, abort, ())$

3.2.3.2 Objets transactionnels et Comportements transactionnels

Objets Transactionnels Soient $o_{TransactionalClass} \in \mathcal{M}_c$ et $o_{TransactionalObject} \in \mathcal{M}_o$ tels que :

- $o_{TransactionalObject} = root(o_{TransactionalClass})$;
- $o_{TransactionalClass} \prec o_{Class}$ et $o_{TransactionalClass} \dashv o_{Class}$;
- $o_{TransactionalObject} \prec o_{Transaction}$ et $o_{TransactionalObject} \dashv o_{TransactionalClass}$.

[3] Une implication entre deux envois de message indique que l'exécution de l'un implique l'exécution de l'autre.

Nous appellerons métaclasse de gestion des objets transactionnels tout objet référencé par $o \in \mathcal{M}_c$ tel que $o \prec o_{TransactionalClass}$. Nous appellerons classe transactionnelle tout objet référencé par $o' \in \mathcal{M}_o$ tel que $o' \prec o_{TransactionalObject}$ et $o' \in \widetilde{\pi}(o_{TransactionalClass})$. Soit $\mathcal{M}_t$ l'ensemble des classes transactionnelles, nous appellerons O_t, l'ensemble des objets transactionnels, le sous-ensemble de O défini de la manière suivante :

$$O_t = \{o \in O \mid \gamma(o) \in \mathcal{M}_t\} \tag{3.2}$$

Comportements Transactionnels Soient $o_{BehaviorClass} \in \mathcal{M}_c$ et $o_{BehaviorObject} \in \mathcal{M}_o$ tels que :

- $o_{BehaviorObject} = root(o_{BehaviorClass})$;
- $o_{BehaviorClass} \prec o_{Class}$ et $o_{BehaviorClass} \dashv o_{Class}$;
- $o_{BehaviorObject} \prec o_{Object}$ et $o_{BehaviorObject} \dashv o_{BehaviorClass}$.

Nous appellerons métaclasse de comportement transactionnel tout objet référencé par $o \in \mathcal{M}_c$ tel que $o \prec o_{BehaviorClass}$. Nous appellerons classe de comportement transactionnel tout objet référencé par $o' \in \mathcal{M}_o$ tel que $o' \prec o_{BehaviorObject}$ et $o' \in \widetilde{\pi}(o_{BehaviorClass})$. Soit $\mathcal{M}_b$ l'ensemble des classes de comportement transactionnel, nous appellerons O_b, l'ensemble des comportements transactionnels, sous-ensemble de O, défini de la manière suivante :

$$O_b = \{o \in O \mid \gamma(o) \in \mathcal{M}_b\} \tag{3.3}$$

Fonction d'association transactionnelle structurelle Nous appellerons fonction d'association transactionnelle structurelle une fonction, notée κ_s, de $\mathcal{M}_b$ dans $\mathcal{M}_t$ associant à chaque classe de comportement transactionnel une classe transactionnelle telle que $\forall o \in \mathcal{M}_b$, $\rho(o) \supseteq \rho(\kappa_s(o))$

Fonction d'association transactionnelle Nous appellerons fonction d'association transactionnelle une fonction, notée κ_t, de O_t dans O_b associant à chaque objet transactionnel un comportement transactionnel telle que :

- $\forall o \in O_t$, $\kappa_s(\gamma(\kappa_t(o)) = \gamma(o)$.
- $\forall o \in O_t$, $\forall s \in \rho(\gamma(o))$, pour tout $a \in L^{fin}(O)$ tel que $a \ll sig(\alpha(\gamma(o))(s))$, on ait $\lambda'(o, s, a) \Rightarrow \lambda'(\kappa_t(o), s, a)$.

3.2.4 Quelques Comportements transactionnels

Nous avons défini précédemment la notion de comportement transactionnel. Nous avons vu que de tels comportements sont associés aux objets transactionnels pour donner à ceux-ci certaines propriétés lorsqu'ils sont mis en œuvre au sein d'une transaction. Il nous semble intéressant de présenter maintenant des comportements transactionnels standard, c'est-à-dire ceux pouvant être utilisés pour développer une application transactionnelle traditionnelle. Nous avons vu dans le premier chapitre que ces comportements sont basés sur les propriétés ACID. Jusqu'à présent, nous nous sommes restreints aux données volatiles et nous n'avons présenté aucun

moyen de définir des contraintes sur ces données. En conséquence, nous nous concentrerons uniquement sur la définition de comportement transactionnels associés aux propriétés AI, celles-ci étant les seules à pouvoir être adaptées à notre modèle tel que nous l'avons présenté jusqu'ici.

3.2.4.1 Atomicité

Commençons par définir un comportement transactionnel permettant de garantir la propriété d'atomicité pour les objets transactionnels qui lui sont associés. Remarquons que, selon notre modèle, ce n'est pas la transaction qui assure la propriété d'atomicité. L'exécution d'une transaction n'est effectivement atomique au sens traditionnel du terme que si tous les objets manipulés par cette transaction sont des objets transactionnels et sont associés à un comportement atomique.

Atomicité et envoi de message La propriété traditionnelle d'atomicité stipule que soit une transaction se termine correctement, et elle a les effets désirés sur les objets manipulés, soit la transaction est interrompue et elle n'a aucun effet sur ces objets. Dans un environnement où les transactions ne contrôlent que les objets persistants, l'atomicité est généralement garantie par un système permettant d'annuler d'une part les mises à jour des objets et, d'autre part, l'ajout ou la suppression d'objets dans la base. En d'autres termes, pour garantir l'atomicité dans un tel environnement, il faut pouvoir annuler les opérations d'écritures dans la base. Dans un environnement plus général où l'on souhaite contrôler de manière transactionnelle tous les comportements de l'application et plus seulement les objets persistants, il faut aussi pouvoir annuler d'autres types d'opérations dites à effet de bord, comme par exemple la gestion des flux d'entrées/sorties. En programmation à objets, toute opération est effectuée par l'envoi d'un message sur un objet entraînant l'exécution d'une méthode. En particulier, la mise à jour d'un objet ou l'exécution d'une opération à effet de bord peut être considérée comme l'exécution d'une méthode sur un objet. Si l'on se situe dans un environnement à métaobjets, la création ou la destruction d'objets est aussi gérée par envois de message sur des métaobjets. Pour garantir la propriété d'atomicité au niveau d'un objet, il suffit donc d'annuler les effets des méthodes exécutées par envoi de message sur cet objet depuis le début de la transaction considérée. Notons que ceci implique que nous considérions que les créations ou destructions d'un objet ne soient pas associées à l'objet mais à sa classe. Dans le cas contraire, la propriété d'atomicité ne pourrait être vérifiée sur l'objet puisqu'il serait impossible d'annuler les créations ou destructions de l'objet. En conséquence, garantir l'atomicité d'une exécution transactionnelle au sens traditionnel du terme impose que les objets impliqués dans l'exécution ainsi que leur classe soient des objets transactionnels associés à un comportement atomique.

Partitionnement des méthodes Nous avons présenté dans le chapitre précédent (voir 2.3.2.9) un sous-ensemble de l'ensemble des méthodes constitué des accesseurs. Ceux-ci sont des méthodes spécifiques permettant d'exploiter les données des objets. Décomposons cet ensemble en deux autres sous-ensembles, celui des accesseurs dédiés à la lecture des données et le second à l'écriture des données. De manière plus formelle, soit $o_{ReadAccessor} \in O$ et $o_{WriteAccessor} \in O$ tels que :

- $o_{ReadAccessor} \prec o_{Accessor}$ et $o_{ReadAccessor} \dashv o_{MetaMethod}$.

- $o_{WriteAccessor} \prec o_{Accessor}$ et $o_{WriteAccessor} \dashv o_{MetaMethod}$.

Nous appellerons accesseur en lecture tout objet $o \in O$ tel que $o \in \widetilde{\pi}(o_{ReadAccessor})$. Nous appellerons accesseur en écriture tout objet $o \in O$ tel que $o \in \widetilde{\pi}(o_{WriteAccessor})$. Enfin, nous supposerons que :

- $\widetilde{\pi}(o_{WriteAccessor}) \cap \widetilde{\pi}(o_{ReadAccessor}) = \{o_{Void}\}$ et
- $\widetilde{\pi}(o_{WriteAccessor}) \cup \widetilde{\pi}(o_{ReadAccessor}) = \widetilde{\pi}(o_{Accessor})$.

En conséquence, garantir l'atomicité d'un objet o implique de pouvoir annuler les effets de toutes les méthodes $m \in \mu_o(o)$ telles que :

- $m \in \widetilde{\pi}(o_{WriteAccessor})$ (1) ou
- $m \in \mathcal{B} \setminus \widetilde{\pi}(o_{Accessor})$ (2).

Notons qu'il n'est pas nécessaire d'annuler les effets des accesseurs en lecture, ceux-ci ne modifiant ni l'état de l'objet ni l'état de l'application par effets de bord.

Pour annuler automatiquement les effets d'une méthode, il est nécessaire soit de connaître la sémantique de la méthode considérée, soit de disposer d'une méthode « inverse ». Alors qu'il est assez simple de déterminer la sémantique des accesseurs en écriture, il semble beaucoup plus difficile de disposer de suffisamment d'informations sémantiques sur les autres méthodes pour permettre leur annulation. Les traitements en cas d'annulation de ces deux types de méthode doivent donc être distincts. Nous les détaillons ci-dessous.

Traitements des accesseurs en écriture Par définition, un accesseur en écriture modifie une partie des données contenues dans les objets instances de la classe au sein de laquelle il est défini. Comme nous l'avons déjà précisé, nous souhaitons adopter une approche typée. Cela implique que l'accesseur en écriture prenne en paramètre un objet dont la classe décrit le type de la partie des données à modifier, donc de l'objet représentant ces données. Par conséquent, permettre l'annulation des accesseurs en écriture peut-être réalisé en copiant les données à modifier, lorsqu'une transaction accède pour la première fois en écriture à l'objet. Par la suite, il suffit de recopier la valeur sauvegardée en cas d'annulation.

Décrivons plus précisément la marche à suivre pour garantir la propriété d'atomicité sur un objet transactionnel accédé par une transaction :

- Lorsqu'une transaction accède pour la première fois en écriture à un objet, il est nécessaire de copier la valeur initiale des données correspondantes et d'associer cette copie à la transaction courante (il peut y avoir plusieurs transactions concurrentes) ainsi qu'à la transaction dans l'environnement de laquelle celle-ci doit etre validée dans le cas où celle-ci n'est déjà pas associée à une copie des mêmes données. Notons que cette dernière opération doit être effectuée récursivement jusqu'à la racine de l'arbre transactionnel correspondant. Dans le cas contraire, si une transaction est validée et que sa transaction mère doit par la suite être annulée, il ne serait pas possible de remettre l'objet concerné dans l'état précédent le début de la transaction.
- Lorsqu'une transaction est annulée sur l'objet courant, celui-ci est remis dans son état précédent le début de la transaction à l'aide des copies associées à cette transaction. Puis, les copies qui ne sont associées à aucune autre transaction sont supprimées.
- Lorsqu'une transaction est validée sur l'objet courant, il suffit de supprimer les copies qui lui sont associées lorsque celles-ci ne sont pas aussi associées à d'autres transactions.

Notons que cet algorithme ne garantit en aucun cas l'isolation des transactions et donc une exécution correcte en cas de concurrence inter-transactions. Cette dernière propriété est dédiée à un comportement transactionnel différent, pouvant néanmoins être combiné à celui-ci, par exemple, en créant une nouvelle métaclasse de comportement transactionnel héritant simultanément des deux métaclasses concernées.

Traitements des autres méthodes Le traitement de ces méthodes implique de disposer des méthodes inverses. Il n'est en effet pas possible, dans la majorité des cas de déduire le comportement à adopter en cas d'annulation à partir du corps des méthodes à annuler. Lors de la déclaration du comportement transactionnel atomique adapté à une classe, l'utilisateur de notre système devra donc, s'il souhaite pouvoir permettre l'annulation d'une ou plusieurs des méthodes décrites dans cette classe fournir les méthodes inverses. Pour permettre l'annulation d'une méthode, le système transactionnel conserve pour chaque transaction une trace simple de ses exécutions. Par la suite, si une transaction est annulée, les méthodes inverses correspondant à la trace lui étant associée sont exécutées. Si une transaction valide, la trace qui lui correspond est associée à sa transaction mère pour garantir la propriété d'atomicité en cas d'annulation de cette dernière. Notons que cette technique n'est pas idéale et ne permet sans doute pas d'annuler tous les types de méthodes. Néanmoins, on peut aisément être convaincu de son utilité par exemple pour des traitements d'ouverture ou de fermeture de fichiers.

Formalisation Soit $o_{AtomicBehavior} \in O$ tel que $o_{AtomicBehavior} \prec o_{BehaviorClass}$ et $o_{AtomicBehavior} \dashv o_{Class}$.

Nous dirons qu'un objet $o \in O$ a un comportement atomique si $\gamma(\kappa_t(o))$ est élément de $\widetilde{\pi}(o_{AtomicBehavior}) \setminus \{o_{Void}\}$.

Considérons l'ensemble, notée $\mathcal{B}_{inv}$ des méthodes $m \in \mathcal{B} \setminus \widetilde{\pi}(o_{Accessor})$ pouvant être inversées[4], nous appellerons fonction d'inversion, notée inv, une fonction de $\mathcal{B}_{inv}$ dans $\mathcal{B}$ associant à chaque méthode une méthode inverse telle que $\forall m \in \mathcal{B}_{inv}$ et $\forall c \in \widetilde{\pi}(o_{AtomicBehavior})$ alors $m \in \mu(\kappa_s(c)) \Rightarrow inv(m) \in \mu(c)$.

Pour terminer cette sous-partie, nous reprenons ci-dessous les algorithmes permettant de garantir l'atomicité d'un objet transactionnel $o \in O$ tel que $\gamma(\kappa_t(o)) \in \widetilde{\pi}(o_{AtomicBehavior}) \setminus \{o_{Void}\}$ par rapport à une transaction $t \in \widetilde{\pi}(o_{Transaction})$. Nous noterons $trace(o,t)$ la trace sur o des méthodes inversibles associées à t, c'est-à-dire la liste des référents de méthodes inversibles ayant été accédées par la transaction t. Pour tout $m \in \widetilde{\pi}(o_{WriteAccessor})$, nous noterons $copie(o,m,t)$, la copie des données contenues dans o correspondant à l'accesseur m et associée à t.

- Envoi de messages. Nous avons vu précédemment que tout envoi de message sur un objet transactionnel impliquait l'envoi du même message sur le comportement transactionnel de cet objet. C'est ce dernier qui nous intéresse. Nous en décrivons ici les propriétés en supposant que le message originellement envoyé à o est de la forme $\lambda'(o,s,a)$. On a alors :

  1. $\lambda'(o,s,a) = \lambda'(\kappa_t(o),s,a)$

[4]Cette propriété peut-être intrinsèque à la méthode ou tout simplement déduite des descriptions de l'utilisateur de notre système.

2. $\lambda'(\kappa_t(o), s, a)$ implique l'exécution de l'algorithme suivant (on suppose ici que $s \notin \{abort, commit, prepare\}$) :
   - Tout d'abord, Soit $m = \alpha(\gamma(o))(s)$:
     (a) si $m \in \widetilde{\pi}(o_{WriteAccessor})$ et si $copie(o, m, t)$ n'existe pas, il faut faire une copie, l'associer à t puis récursivement pour tout k à $getCommit^k(t)$, $getCommit^k$ étant la composée de $getCommit$ avec elle-même k fois.
     (b) si $m \in \mathcal{B}_{inv} \setminus \widetilde{\pi}(o_{Accessor})$ m est ajoutée à la liste $trace(o, t)$.
     (c) dans les autres cas, il n'y a rien faire.
   - Puis, le résultat de $\lambda(o, m, a)$ est retourné.

- Annulation. L'envoi du message d'annulation correspond à l'envoi du message $\lambda'(\kappa_t(o), s, a)$ avec $s = abort$. Celui-ci implique l'exécution de l'algorithme suivant :

  1. Pout tout $o' \in O$ tel qu'il existe $m \in o_{WriteAccessor}$ tel que $o' = copie(o, m, t)$, il faut remplacer la partie des données de o correspondant à m par o'. S'il n'existe pas $t' \in \widetilde{\pi}(o_{Transaction})$ tel que $t' \neq t$ et $o' = copie(o, m, t')$ il faut alors supprimer o'.
  2. Pour tout $m \in trace(o, t)$ il faut exécuter la méthode correspondant à $\lambda(o, inv(m), ())$ puis enlever m de la liste $trace(o, t)$.

- Préparation. L'envoi du message de préparation correspond à l'envoi du message $\lambda'(\kappa_t(o), s, a)$ avec $s = prepare$. Comme il n'est posée aucune contrainte sur l'objet et que nous nous plaçons actuellement dans un environnement volatile, cette opération retourne dans tous les cas un acquittement positif.
- Validation. L'envoi du message de validation correspond à l'envoi du message $\lambda'(\kappa_t(o), s, a)$ avec $s = commit$. Celui-ci implique l'exécution de l'algorithme suivant :

  1. Pout tout $o' \in O$ tel qu'il existe $m \in o_{WriteAccessor}$ tel que $o' = copie(o, m, t)$, s'il n'existe pas $t' \in \widetilde{\pi}(o_{Transaction})$ tel que $t' \neq t$ et $o' = copie(o, m, t')$, il faut supprimer o'.
  2. Pour tout $m \in trace(o, t)$, enlever m de la liste $trace(o, t)$.

3.2.4.2 Contrôle de concurrence par verrouillage

Pour terminer cette partie, attachons-nous à définir un comportement transactionnel dédié au contrôle de concurrence.

Remarquons comme précédemment que, selon notre modèle, ce n'est pas la transaction qui assure la propriété d'isolation. L'exécution d'une transaction n'est effectivement isolée au sens traditionnel du terme que si tous les objets manipulés par cette transaction sont associés à un comportement transactionnel garantissant l'isolation de la transaction sur ces objets.

Isolation et envoi de messages Nous considérons dans ce mémoire que l'envoi de message est le medium de communication unique entre objets. De plus, nous souhaitons que les algorithmes de gestion transactionnelle soient délégués et distribués au sein des objets transactionnels. Dans ce cadre, garantir l'isolation des transactions peut être réalisé en adaptant les algorithmes de gestion de concurrence décrits dans le premier chapitre pour notre système.

Nous avons choisi ici d'étudier l'algorithme de verrouillage en 2 phases. Pour mémoire, cet algorithme est basé sur la décomposition d'une transaction en opérations atomiques de lecture ou d'écriture de données et suit les règles suivantes :

- Avant chaque lecture ou écriture d'une donnée, toute transaction doit poser respectivement un verrou en lecture ou un verrou en écriture sur cette donnée.
- Il est possible de poser un verrou en lecture sur une donnée si celle-ci ne possède pas déjà un verrou en écriture.
- Il est possible de poser un verrou en écriture sur une donnée si celle-ci ne possède pas déjà ni un verrou en lecture, ni un verrou en écriture.
- Enfin, dès qu'une transaction a relâché un verrou, il ne lui est plus possible d'acquérir de nouveaux verrous.

Il est possible de retranscrire ces mêmes règles au sein de notre système au niveau des accesseurs qui représentent la décomposition d'une transaction en opérations de lecture et d'écriture. Comme précédemment, cela implique que le contrôle de concurrence dédié à la création ou de la destruction des objets doit être délégué aux classes considérées comme des objets transactionnels. Dans cette partie, nous considérons et nous séparons comme précédemment les accesseurs en lecture éléments de $\widetilde{\pi}(o_{ReadAccessor})$ et les accesseurs en écriture éléments de $\widetilde{\pi}(o_{WriteAccessor})$

Verrouillage en 2 phases et transactions imbriquées Comme nous l'avons vu dans le chapitre 1, l'algorithme de verrouillage à 2 phases doit être modifié lorsqu'il est mis en œuvre dans le cadre d'un modèle de transactions imbriquées. Pour mémoire, lorsque seules les transactions feuilles accèdent aux données, il doit être de la forme suivante :

- Avant chaque lecture ou écriture d'une donnée, une sous-transaction doit poser respectivement un verrou en lecture ou un verrou en écriture sur cette même donnée.
- Il est possible de poser un verrou en lecture sur une donnée si celle-ci ne possède pas déjà de verrou en écriture ou si toutes les transactions qui possèdent un verrou en écriture sur la donnée sont des ancêtres de la sous-transaction.
- Il est possible de poser un verrou en écriture sur une donnée si celle-ci ne possède pas déjà ni de verrou en lecture, ni de verrou en écriture ou si toutes les transactions qui possèdent un verrou en écriture ou en lecture sur la donnée sont des ancêtres de la sous-transaction.
- Lorsqu'une sous-transaction est validée, tous ses verrous sont anti-hérités par sa transaction mère. En d'autres termes, sa transaction mère acquiert ces verrous par héritage inverse.
- Lorsqu'une sous-transaction est annulée, tous ses verrous sont relâchés.

Notre modèle transactionnel est plus général que celui des transactions imbriquées. En particulier, nous autorisons toutes les formes possibles de concurrence entre transactions et sous-transactions. Utiliser le même algorithme que celui présenté ci-dessus pour garantir la cohérence des données de notre système résulterait en de nombreuses incohérences entre transactions d'une même hiérarchie. En effet, comme les sous-transactions peuvent obtenir un verrou, par exemple en écriture, si l'un au moins de leurs ancêtres en possède un, le résultat de l'exécution d'une transaction en concurrence avec un de ces ancêtres peut être incohérent. Une première solution à ce problème peut être tout simplement de supprimer cette propriété. Néanmoins, ce n'est pas une bonne solution car cela peut entrainer des abandons en cascade comme nous l'avons vu dans le chapitre 1. Pour résoudre ce problème, nous introduisons deux

nouveaux types de verrous que sont les verrous d'héritage en lecture et les verrous d'héritage en écriture. L'algorithme de verrouillage suit alors les règles suivantes :

- Avant chaque lecture ou écriture d'une donnée, toute transaction doit poser respectivement un verrou en lecture ou un verrou en écriture sur cette même donnée.
- Il est possible de poser un verrou en lecture sur une donnée si et seulement si l'une, au moins, des propriétés énumérées ci-dessous est vérifiée :

  1. la transaction possède un verrou en écriture.
  2. aucun verrou en écriture n'est posé sur la donnée et tous les verrous d'héritage en écritures sont possédés par des ancêtres de la transaction ou par elle-même.

- Il est possible de poser un verrou en écriture sur une donnée si et seulement si l'une, au moins, des propriétés énumérées ci-dessous est vérifiée :

  1. la transaction considérée possède l'unique verrou en lecture sur la donnée, aucun verrou en écriture n'est posé sur la donnée et tous les verrous d'héritage en lecture ou en écriture sont possédés par des ancêtres de la transaction ou par elle-même.
  2. aucun verrou en écriture ou en lecture n'est posé sur la donnée et tous les verrous d'héritage en lecture ou en écriture sont possédés par des ancêtres de la transaction ou par elle-même.

- Lorsqu'une transaction est validée, tous ses verrous d'héritage sont anti-hérités[5] par sa transaction mère. De plus, tous ses verrous en lecture ou en écriture sont transformés en verrous d'héritage correspondant puis anti-hérités par sa transaction mère.
- Lorsqu'une transaction est annulée, tous ses verrous sont relâchés.

Cet algorithme garantit comme précédemment la propriété de non-abandon en cascade grâce aux règles posées sur les verrous d'héritage. De plus, la cohérence de l'exécution en concurrence de transactions de la même hiérarchie est garantie.

A propros des verrous mortels La modification de l'algorithme de verrouillage présenté précédemment nécessite de modifier aussi les algorithmes de détection de verrous mortels. Nous avons vu dans le chapitre 1 deux types d'algorithmes sur ce thème. Le premier construit le graphe des attentes (Wait-For-Graph) et détecte un verrou mortel lorsqu'il y a un circuit dans le graphe. Le second impose un délai de garde à chaque transaction au dela duquel la transaction est considérée comme faisant partie d'un verrou mortel. Nous avons choisi d'étudier et d'adopter la première approche.

Commençons par déterminer les objets pouvant participer à la construction du graphe des attentes. Dans notre modèle, les transactions n'ont qu'un rôle de structuration de l'exécution. En particulier, aucun comportement de contrôle de concurrence ne leur est associé. Il n'est donc pas possible de mettre en œuvre la construction d'un graphe des attentes au sein d'une transaction car cela compromettrait la généralité et l'ouverture de notre modèle. Il en est de même des objets transactionnels qui ne sont associés que temporairement avec un comportement transactionnel particulier. Il nous reste donc deux possibilités, soit définir un nouvel objet chargé de construire le graphe des attentes et de détecter les verrous mortels, soit de

[5]Ce sont les parents qui obtiennent les verrous de leurs enfants. On parle aussi d'héritage ascendant.

distribuer cette tâche au niveau des comportements transactionnels utilisant un algorithme pouvant induire des verrous mortels. Dans le contexte particulier de nos modélisations, la deuxième approche nous semble la plus intéressante. En effet, si l'utilisateur décide d'ajouter de nouveaux comportements transactionnels pouvant induire des verrous mortels, la détection de ces derniers doit pouvoir être assez facilement mise en œuvre. Or, dédier cette détection à un objet particulier fige les techniques qui lui sont associées et implique de poser des contraintes fortes sur les types de comportement transactionnels qu'il est possible de définir. De plus, dans le cadre de l'extension de notre modèle aux données distribuées, il est plus logique de mettre en place des algorithmes intrinsèquement distribués.

A présent, Déterminons quels sont les différents risques d'attente entre deux transactions.

- Tout d'abord, comme nous l'avons déjà vu, une transaction attend la terminaison de ses sous-transactions. Comme les verrous ne sont relâchés qu'à la fin d'une transaction, il est possible que ce type d'attente génère un verrou mortel. En effet, prenons l'exemple de trois transactions t_1, t_2 et t_3 telles que t_1 soit sous-transaction de t_2. Soient deux données a et b telles que t_3 possède un verrou exclusif sur a et t_2 un verrou exclusif sur b. Supposons de plus que t_3 souhaite écrire b et que t_1 souhaite écrire a. On a alors t_1 qui attend la levée du verrou posé sur b c'est-à-dire la fin de t_3. Or t_3 attend la levée du verrou posé sur a, c'est-à-dire la fin de t_2, celle-ci devant elle-même attendre la fin de t_1 pour terminer. On obtient un verrou mortel qui n'aurait pas été formé si t_1 n'était pas une sous-transaction de t_2.
- Lorsqu'une transaction veut lire une donnée, s'il existe sur cette donnée un verrou en écriture ou des verrous d'héritage en écriture non possédés par ses ancêtres, alors elle doit alors attendre la levée de tous ces verrous pour continuer.
- Lorsqu'une transaction veut écrire une donnée, s'il existe sur cette donnée un verrou en écriture ou en lecture ou bien des verrous d'héritage non possédés par ses ancêtres, alors elle doit alors attendre la levée de tous ces verrous pour continuer.

Nous avons choisi de distribuer le contrôle des verrous mortels sur les différents comportements transactionnels utilisant un contrôle de concurrence pouvant induire ces verrous. Il est donc nécessaire que tous ces comportements puissent recevoir les messages relatifs à l'algorithme de détection. Dans notre modèle, nous restreignons cet ensemble à deux messages, $waitForOn$ et $waitFor$. Le premier, directement lié au comportement transactionnel le recevant, doit permettre de connaître si une transaction t_1 est en attente d'une autre transaction t_2 par rapport à l'objet transactionnel associé au comportement. Le second message, plus général, doit permettre de connaître si une transaction t_1 attend une autre transaction t_2 quelle que soit la raison de cette attente. Comme ce dernier message ne dépend pas du comportement transactionnel ni de l'objet transactionnel, on pourrait définir la méthode correspondante comme une méthode statique en C++. Au sein de notre modèle, nous considérons que ce message peut être envoyé à une classe, racine d'héritage associée à la métaclasse de comportement à partir de laquelle tous les comportements transactionnels pouvant induire des verrous mortels doivent être instanciés.

Sans rentrer dans les détails, ce qui sera fait plus loin, l'algorithme de détection de verrous mortels adaptés à notre système de verrouillage et correspondant à chacun de ces messages est alors de la forme suivante en supposant que les méthodes associées à ces messages prennent en arguments t_1, t_2 et un ensemble e_1 de transactions déjà traitées.

- message $waitForOn$. Tout d'abord, on forme l'ensemble des transactions en concurrence

avec t_1 sur l'objet considéré et devant être terminées pour permettre la reprise de t_1. t_1 est alors en attente sur t_2, si et seulement si t_2 fait partie de cet ensemble ou si l'une des transactions non traitées de cet ensemble est en attente de t_2 (message $waitFor$).
- message $waitFor$. Ce message résulte en l'exécution des trois étapes suivantes. Tout d'abord, on rajoute t_1 à la liste des transactions traitées. Puis, on forme l'ensemble des sous-transactions non traitées devant être terminées pour pouvoir terminer t_1. t_1 est alors en attente sur t_2, si et seulement si t_2 fait partie de cet ensemble ou si l'une des transactions de cet ensemble est en attente de t_2 (message $waitFor$). Puis, on forme l'ensemble des objets transactionnels pouvant répondre au message $waitForOn$ accédés par t_1. t_1 est alors en attente sur t_2, si et seulement si t_1 est en attente de t_2 sur l'un des objets de cet ensemble.

Finalement, pour détecter les verrous mortels, nous utilisons l'algorithme suivant. Lorsqu'une transaction t_1 ne peut poser un verrou sur un objet transactionnel, nous considérons l'ensemble des transactions devant être terminées pour permettre la pose du verrou. Si l'une des transactions de cet ensemble est en attente sur t_1 (message $waitFor$) alors l'attente de t_1 provoquerait un verrou mortel. Dans ce cas, la transaction t_1 est annulée. Notons que cette annulation peut ne pas être immédiate dans le cas où t_1 doit attendre la fin d'une sous-transaction pour annuler. Néanmoins, cette nouvelle attente ne peut induire un nouveau verrou mortel. En effet, les sous-transactions concernées ne peuvent être en attente de la terminaison de leur transaction mère à la suite d'une demande de verrou. Dans le cas contraire, un verrou mortel aurait précédemment été détecté et résolu par l'annulation de la transaction concernée.

Formalisation Soit $o_{LockingBehavior}$ et $o_{lockingObject}$ deux éléments de O tels que :

- $o_{LockingBehavior} \prec o_{BehaviorClass}$ et $o_{LockingBehavior} \dashv o_{Class}$
- $o_{LockingObject} \prec o_{BehaviorObject}$ et $o_{LockingObject} \dashv o_{LockingBehavior}$
- $root(o_{LockingBehavior}) = o_{LockingObject}$
- $\rho(o_{LockingBehavior}) \supset \{waitFor\}$
- $\rho(o_{LockingObject}) \supset \{waitForOn\}$

Nous dirons qu'un objet $o \in O$ à un comportement transactionnel par verrouillage si et seulement si $\kappa_t(o) \in \widetilde{\pi}(o_{LockingObject}) \setminus \{o_{Void}\}$

Nous noterons :

- o' un comportement transactionnel
- o un objet transactionnel
- t une transaction
- $rConcurrentWith(o', t)$ l'ensemble des transactions en concurrence avec t pour lire l'objet transactionnel auquel o' est associé, c'est-à-dire la liste des transactions devant être terminées pour permettre à t de lire cet objet.
- $wConcurrentWith(o', t)$, l'ensemble des transactions en concurrence avec t pour modifier l'objet transactionnel auquel o' est associé, c'est-à-dire la liste des transactions devant être terminées pour permettre à t de modifier cet objet.
- $concurrentWith(o', t) = rConcurrentWith(o', t) \cup wConcurrentWith(o', t)$.
- $rLock(o')$ l'ensemble des transactions ayant un verrou en lecture sur l'objet transactionnel auquel o' est associé.
- $irLock(o')$ l'ensemble des transactions ayant un verrou d'héritage en lecture sur l'objet transactionnel auquel o' est associé.

- $iwLock(o')$ l'ensemble des transactions ayant un verrou d'héritage en écriture sur l'objet transactionnel auquel o' est associé.
- $rLock(o')$ la transaction, si elle existe, ayant un verrou en écriture sur l'objet transactionnel auquel o' est associé.

Notons que nous ne permettons pas le verrouillage d'une partie d'un objet mais uniquement le verrouillage d'un objet dans son ensemble. Reprenons maintenant les algorithmes introduits précédemment permettant de garantir la cohérence, lors d'accès concurrents, d'un objet transactionnel $o \in O$, tel que $\kappa_t(o) = o' \in \widetilde{\pi}(o_{LockingObject}) \setminus \{o_{Void}\}$. Nous avons vu précédemment que tout envoi de message sur un objet transactionnel impliquait l'envoi du même message sur le comportement transactionnel de cet objet. C'est ce dernier qui nous intéresse. Nous en décrivons ici les propriétés en supposant que le message originellement envoyé à o est de la forme $\lambda'(o, s, a)$. Nous appellerons m la méthode associée à l'envoi de message telle que $m = \alpha(\gamma(o))(s)$, et t la transaction courante. Diverses attitudes sont à adopter en fonction du message s reçu par o. :

- Dans tous les cas, on a $\lambda'(o, s, a) = \lambda'(o', s, a)$
- Si $s \notin \{abort, prepare, commit\}$ et $m \in \mathcal{B} \setminus \widetilde{\pi}(o_{Accessor})$, il suffit de renvoyer le résultat de $\lambda(o, m, a)$.
- Si $m \in \widetilde{\pi}(o_{readAccessor})$ il faut alors prendre en compte les cas suivants qui sont exclusifs.

  1. Si $t \in rLock(o')$ ou $t = wLock(o')$ alors le résultat de $\lambda(o, m, a)$ est retourné.
  2. Si $wLock(o') = o_{Void}$ et si $\forall t' \in iwLock(o')$, $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ alors t est ajoutée dans l'ensemble $rLock(o')$ et le résultat de $\lambda(o, m, a)$ est retourné.
  3. Dans tous les autres cas, il faut construire l'ensemble e des transactions en concurrence avec t sur o puis vérifier que mettre t en attente n'induise pas un verrou mortel. Si c'est le cas t est annulée par l'envoi de message $\lambda'(t, abort, ())$. Sinon, l'ensemble $concurrentWith(o', t)$ est rendu égal à e et t est mise en attente en envoyant le message $\lambda'(t, wait, ())$. Lorsque t est reprise, les différents cas énumérés ici sont à nouveau examinés.

- Si $m \in \widetilde{\pi}(o_{writeAccessor})$ il faut alors prendre en compte les cas suivants qui sont exclusifs.

  1. Si $t = wLock(o')$ alors le résultat de $\lambda(o, m, a)$ est renvoyé.
  2. Si $wLock(o') = o_{Void}$, si $rLock(o') = \emptyset$, et si $\forall t' \in iwLock(o') \cup irLock(o')$, $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ alors $wLock(o')$ est rendu égal à t et le résultat de $\lambda(o, m, a)$ est renvoyé.
  3. Dans tous les autres cas, il faut construire l'ensemble e des transactions en concurrence avec t sur o puis vérifier que mettre t en attente n'induise pas un verrou mortel. Si c'est le cas, t est annulée par l'envoi de message $\lambda'(t, abort, ())$. Sinon $concurrentWith(o', t)$ est rendu égal à e et t est mise en attente en envoyant le message $\lambda'(t, wait, ())$. Lorsque t est reprise, les différents cas énumérés ici sont à nouveau examinés.

- Annulation. Cela correspond au cas $s = abort$. Tout d'abord, il faut supprimer tous les verrous possédés par la transaction, c'est-à-dire d'une part enlever t des ensembles $rLock(o')$, $irLock(o')$ et $iwLock(o')$ et, d'autre part, si $wLock(o') = t$, rendre $wLock(o')$ égal à o_{Void}. Il faut ensuite supprimer l'ensemble $concurrentWith(o', t)$. Puis pour toute transaction t' distincte de t.

1. Si $concurrentWith(o', t') = \{t\}$, il faut :
   - supprimer l'ensemble $concurrentWith(o', t')$;
   - et redémarrer t' en envoyant le message $\lambda'(t', resume, ())$.
2. Si $concurrentWith(o', t') \supset \{t\}$, il faut :
   - enlever t de l'ensemble $concurrentWith(o', t')$.

- Préparation. Cela correspond au cas $s = prepare$. Dans un environnement volatile, cette opération renvoie toujours un acquittement positif.
- Validation. Cela correspond au cas $s = commit$. Tout d'abord il est nécessaire que la transaction mère de t, si elle existe, anti-hérite des verrous pris par t en suivant les règles. Puis il faut recalculer les ensembles de dépendances inter-transactions sur o. Ceci est précisé par les règles suivantes :

1. Si $t \in rLock(o')$ et $wLock(o') = o_{Void}$, t est supprimée de l'ensemble $rLock(o')$
   (a) si $getCommit(t)$ existe, elle est rajoutée dans l'ensemble $irLock(o')$. De plus, pour toute transaction t' distincte de t :
      i. Si $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ et si $wConcurrentWith(o', t') = \{t\}$, il faut supprimer $wConcurrentWith(o', t')$ et redémarrer t' en envoyant le message $\lambda'(t', resume, ())$
      ii. Si $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ et si $wConcurrentWith(o', t') \supset \{t\}$, il faut supprimer t de $wConcurrentWith(o', t')$.
      iii. Si $wConcurrentWith(o', t') \supseteq \{t\}$, il faut alors supprimer t de l'ensemble $wConcurrentWith(o', t')$ et y rajouter $getCommit(t)$.
   (b) Sinon pour toute transaction t' distincte de t :
      i. Si $wConcurrentWith(o', t') = \{t\}$, il faut supprimer $wConcurrentWith(o', t')$ et redémarrer t' en envoyant le message suivant : $\lambda'(t', resume, ())$
      ii. Si $wConcurrentWith(o', t') \supset \{t\}$, t doit etre alors supprimée de l'ensemble $wConcurrentWith(o', t')$.
2. Si $t = wLock(o')$, $wLock(o')$ est rendu égal à o_{Void} et t est supprimée de l'ensemble $rLock(o')$.
   (a) si $getCommit(t)$ existe, il faut la rajouter dans $iwLock(o')$. De plus, pour toute transaction t distincte de t :
      i. Si $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ et si $concurrentWith(o', t') = \{t\}$, il faut supprimer $concurrentWith(o', t')$ et redémarrer t' en envoyant le message $\lambda'(t', resume, ())$
      ii. Si $\exists k \in \mathbb{N}^*$ tel que $getCommit^k(t) = t'$ et si $concurrentWith(o', t') \supset \{t\}$, il faut supprimer t de $concurrentWith(o', t')$.
      iii. Si $concurrentWith(o, t') \supseteq \{t\}$, il faut supprimer t de $concurrentWith(o', t')$ et rajouter $getCommit(t)$ à la fois dans l'ensemble $rConcurrentWith(o', t')$ et dans l'ensemble $wConcurrentWith(o', t')$.
   (b) Sinon pour toute transaction t' distincte de t :
      i. Si $concurrentWith(o', t') = \{t\}$, il faut supprimer $wConcurrentWith(o, t')$ et redémarrer t' en envoyant le message suivant : $\lambda'(t', resume, ())$
      ii. Si $concurrentWith(o', t') \supset \{t\}$, il faut supprimer t de $concurrentWith(o', t')$.

3. Si $t \in irLock(o')$, il faut supprimer t de $irLock(o')$ puis suivre les règles définies au point 1.
4. Si $t \in iwLock(o')$, il faut supprimer t de $iwLock(o')$ puis suivre les règles définies au point 2.

Pour terminer cette partie, décrivons l'algorithme de détection des verrous mortels. Nous nous plaçons dans le cadre d'une lecture ou d'une écriture d'un objet o par une transaction t. Nous supposons qu'il n'est pas possible de poser le verrou correspondant et que l'ensemble e des transactions en concurrence avec t a été déterminé (cas 3 de l'algorithme de lecture ou d'écriture d'un objet). Nous supposerons de plus qu'il existe un ensemble e', vide au début de l'algorithme de détection puis contenant les transactions contrôlées en cours de l'algorithme. Nous supposerons que le message $waitFor$ peut être envoyé à $o_{lockingObject}$ et que le message $waitForOn$ peut être envoyé à tous les objets $o' \in \tilde{\pi}(o_{lockingObject})$. Dans ce cas, détecter l'éventuel verrou mortel induit par l'attente de t sur o est réduit à envoyer pour tout $t' \in e$ le message $\lambda'(o_{LockingObject}, waitFor, (t', t, e'))$ en supposant que les algorithmes exécutés à la suite d'un message $waitFor$ ou $waitForOn$ sont tels que décrits ci-dessous. Si le résultat de ce message est o_{true}, alors l'attente de la transaction induit un verrou mortel, sinon il est possible de la mettre en attente.

- Message $\lambda'(o_{LockingObject}, waitFor, (t', t, e'))$.

1. Pour commencer, t' est ajouté à e'
2. soit $e_t = getSubs(t') \setminus e'$.
   (a) Si $t' \in e_t$, il faut renvoyer o_{true}.
   (b) Pour tout $t'' \in e_t$, il faut envoyer le message $\lambda'(o_{LockingObject}, waitFor, (t'', t, e'))$. Si l'un de ces messages a pour résultat o_{true}, il faut renvoyer o_{true} sinon renvoyer o_{false} .
3. soit $e_o = getObjects(t') \cap \tilde{\pi}(o_{LockingObject})$. Pour tout $o' \in e_o$ il faut envoyer le message $\lambda'(o', waitForOn, (t', t, e'))$. Si l'un de ces messages a pour résultat o_{true}, il faut renvoyer o_{true}, sinon renvoyer o_{false} .

- Message $\lambda'(o', waitForOn, (t', t, e'))$. Soit $e_c = concurrentWith(o', t') \setminus e'$.

1. Si $t \in e_c$ il faut renvoyer o_{true}
2. Pour tout $t'' \in e_c$, il faut envoyer le message $\lambda'(o_{LockingObject}, waitFor, (t'', t, e'))$. Si l'un de ces messages a pour résultat o_{true} il faut alors renvoyer o_{true} sinon renvoyer o_{false} .

3.3 Relais et requêtes

L'introduction des relais au sein de notre modélisation doit permettre aux utilisateurs, d'une part, de rendre les données de leurs applications persistantes et, d'autre part, d'exploiter des sources de données existantes éventuellement hétérogènes. Les relais sont donc des entités clés pour la résolution de l'hétérogénéité de stockage et de l'hétérogénéité d'accès, dans le sens où ils doivent permettre de gérer des données persistantes provenant éventuellement de

plusieurs sources de données hétérogènes et ce de manière transparente et selon les besoins de l'utilisateur.

Nous supposerons que l'accès aux données persistantes, stockées au sein de systèmes externes au langage, n'est possible qu'à l'aide des langages de requêtes associés à chaque système.

- Pour résoudre l'hétérogénéité de stockage des données, il faut disposer d'un langage de requêtes capable d'exploiter de manière uniforme toutes les données persistantes. En d'autres termes, il faut que notre système dispose d'un module permettant, d'une part, la traduction des requêtes exprimées dans le langage du système en requêtes exprimées suivant les langages des sources de données, et, d'autre part, la reconstruction des données, résultats des requêtes exprimées dans les langages des sources de données, en objets de notre système.
- Pour résoudre l'hétérogénéité d'accès, il faut que le langage de requêtes de notre système puisse être aussi utilisé pour exploiter les données non persistantes. De plus, le module de traitement des requêtes doit être suffisamment ouvert pour permettre à l'utilisateur de notre système de le spécialiser pour intégrer de nouvelles sources de données.

Nous supposerons aussi que l'accès aux données persistantes n'est possible que dans un environnement transactionnel.

- Pour résoudre l'hétérogénéité de stockage, il doit être possible d'émettre de manière homogène des requêtes transactionnelles au dessus de diverses sources de données dont les systèmes transactionnels peuvent être hétérogènes.
- Pour résoudre l'hétérogénéité d'accès, il faut que le système transactionnel propre au données persistantes de notre système s'intègre de manière transparente au système transactionnel défini précédemment pour les données volatiles. En d'autres termes, cela implique que les relais, devant être considérés comme des objets transactionnels standards doivent pouvoir être associés à des comportements transactionnels. De plus, l'utilisateur de notre système ne doit pas avoir à gérer explicitement les systèmes transactionnels des sources de données qu'il utilise, cette tâche devant être déléguée au module transactionnel du système. Enfin, comme précédemment ce module doit être suffisament spécialisable pour permettre la prise en compte de nouvelles sources de données. Notons que cette propriété découle principalement de la mise en œuvre d'un protocole à métobjets.

Avant de décrire et de formaliser le concept de relai, nous présentons un système de requête en environnement volatile.

3.3.1 Requêtes en environnement volatile

3.3.1.1 Introduction

Dans cette sous-partie, nous ne considérons que les données volatiles et nous cherchons à définir un système de gestion de requêtes sur ces données.

Nous considérons qu'une requête peut être représentée par deux chaînes de caractères qui spécifient respectivement :

- une expression de restriction basée sur une expression conditionnelle ;
- une expression de projection.

Nous considérons de plus que l'exécution d'une requête correspond à l'envoi d'un message *select* à un objet non instance terminale et que le résultat de la requête contient la projection, calculée en fonction de la clause de projection associée à la requête, de l'ensemble des instances directes ou indirectes de cet objet vérifiant la restriction associée à la requête. Notons que cette approche est très restrictive par rapport aux langages de requêtes standard tels qu'OQL ou SQL. En particulier, notre approche ne permet pas l'expression de jointures entre plusieurs classes. Nous avons fait ce choix pour plusieurs raisons :

- Le concept de jointure a été introduit dans les langages de requêtes associés au métamodèle relationnel pour permettre l'expression de chemins d'accès entre des tables. Au sein d'un langage à objets, ces liens peuvent être déduits, dans la grande majorité des cas, des liens de référencement entre objets.
- L'implantation du concept de jointure ainsi que son optimisation est très complexe. De plus, l'exécution d'une jointure est coûteuse en termes de temps de calcul.
- Notre approche nous semble bien adaptée à l'esprit de la programmation par objets. En effet, une requête est avant tout un filtrage d'un ensemble d'objets. Dans un langage à objets, les ensembles d'objets les plus souvent manipulés sont les extensions des classes, celles-ci étant gérés par les classes. Il nous semble donc intéressant de dédier aux classes le traitement des requêtes portant sur leur extension. De plus, de cette manière, le traitement des requêtes peut être spécialisé pour chaque classe ou tout du moins pour chaque type de métaobjet, ce qui nous permet de conserver une approche ouverte et paramétrable,

3.3.1.2 Formalisation

Nous détaillons ci-dessous une extension à notre modèle formel prenant en compte la gestion des requêtes sur les données volatiles.

Requêtes Soit o_{Query} telle que $o_{Query} \prec o_{Object}$ et $o_{Query} \dashv o_{Class}$. Nous appellerons requête tout objet $o \in \widetilde{\pi}(o_{Query})$.

Considérons les fonctions, nommées $where$ et $proj$, de $\widetilde{\pi}(o_{Query})$ dans $\mathcal{S}$ associant à chaque requête sa clause de restriction et sa clause de projection.

De plus, Nous supposerons que $\{applyWhere, applyProj\} \subseteq \rho(o_{Query})$ tel que :

- $ret(\alpha(o_{Query})(applyWhere)) = o_{Boolean}$;
- $sig(\alpha(o_{Query})(applyWhere)) = (o_{Object})$;
- $ret(\alpha(o_{Query})(applyProj)) = o_{Object}$;
- et $sig(\alpha(o_{Query})(applyProj)) = (o_{Object})$.

Ensembles résultats de requêtes Soit o_{qSet} et $o_{qSetOfObject}$ deux éléments de O tels que :

- $o_{qSet} \prec o_{Set}$ et $o_{qSet} \dashv o_{Class}$
- $o_{qSetOfObjects} \prec o_{Type}$ et $o_{qSetOfObjects} \dashv o_{qSet}$.
- $\{getElement\} \subseteq \rho(o_{qSetOfObjects})$

Considérons de plus la fonction nommée $query$ associant à chaque $o \in \widetilde{\pi}(o_{qSetOfObjects})$ une requête $r \in \widetilde{\pi}(o_{Query})$.

Considérons la fonction nommée $elements$ associant à chaque $o \in \widetilde{\pi}(o_{qSetOfObjects})$ l'ensemble des objets qu'il contient.

Métaclasse de requêtes Soient $o_{QueryAbleClass}$ et $o_{QueryAbleObject}$ deux éléments de O tels que :

- $o_{QueryAbleObject} = root(o_{QueryAbleClass})$;
- $o_{QueryAbleClass} \prec o_{Class}$ et $o_{QueryAbleClass} \dashv o_{Class}$;
- $o_{QueryAbleObject} \prec o_{Object}$ et $o_{QueryAbleObject} \dashv o_{QueryAbleClass}$;
- $\{select\} \subseteq \rho(o_{QueryAbleClass})$.
- $ret(\alpha(o_{QueryAbleClass})(select)) = o_{qSetOfObject}$ et $sig(\alpha(o_{QueryAbleClass})(select)) = (o_{Query})$.

Algorithme de traitement des requêtes L'algorithme de traitement des requêtes correspond d'une part à l'exécution de la méthode induite par l'envoi d'un message $\lambda'(c, select, (r))$ avec c la classe sur laquelle est émise la requête r et d'autre part à l'exploitation de l'ensemble résultant. Il suit les règles décrites ci-dessous :

- $\lambda'(c, select, (r)) = e$ tel que
  1. $e \in \widetilde{\pi}(o_{qSetOfObjects})$, $query(e) = r$
  2. $elements(e) = \{o \in \widetilde{\pi}(c) \mid \lambda'(r, applyWhere, (o)) = o_{true}\}$
- $\lambda'(e, getElement, ()) = o$ tel que $\exists o' \in elements(e), \lambda'(query(e), applyProj, (o')) = o$.

Nous appellerons dans la suite $resultElements(e)$, l'ensemble des objets résultats de la requête associée à $e \in \widetilde{\pi}(o_{qSetOfObjects})$. En d'autres termes $resultElements(e)$ est l'ensemble des $o \in O$ tel que $\exists o' \in elements(e), \lambda'(query(e), applyProj, (o')) = o$. Notons que l'exécution d'une requête est en deux temps. Tout d'abord, on filtre l'extension de c à l'aide de la clause de restriction de la requête r. Puis, lors de l'exploitation du résultat, on projette, lorsque cela est nécessaire l'objet obtenu en fonction de la clause de projection de r. Notons aussi que, dans le cas où les objets résultants de la restriction sont modifiés entre le moment où l'utilisateur reçoit l'ensemble résultant et le moment où il traite cet ensemble, notre approche peut conduire à des résultats différents d'une approche classique qui impose l'exécution de la projection avant de renvoyer le résultat. Notre approche est plus dynamique et peut permettre de prendre en compte les modifications des objets modifiés sans effectuer de nouvelles requêtes. En contrepartie, les objets résultants de la restriction peuvent ne plus la vérifier lors de l'exécution de la projection ce qui peut induire certaines incohérences.

3.3.2 Relais et données persistantes.

3.3.2.1 Sources de données.

Pour exploiter des sources de données externes au langage, il est nécessaire de décrire la manière d'y accéder, ce qui inclut comme nous l'avons déjà présenté :

- la gestion des requêtes
- la mise à jour des données
- l'exploitation du système transactionnel de la source de données concernée lorsqu'il existe.

Dans cette sous-partie, nous formalisons la représentation de telles sources de données. Nous supposerons que toute source stocke ces données structurées sous la forme d'entité de type n-uplet. De plus, nous supposerons que chaque entité stockée peut-être associée à un identifiant unique. Dans ce cadre, nous introduisons les objets suivants.

Soit $o_{ExternalTransaction}$ elément de O tel que :

- $o_{ExternalTransaction} \prec o_{Transaction}$;
- et $o_{ExternalTransaction} \dashv o_{Class}$.

Soient $o_{ExternalSource}$ élément de O tel que :

- $o_{ExternalSource} \prec o_{Object}$ et $o_{ExternalSource} \dashv o_{Class}$.
- $\{query, getObject, updateObject, newObject, deleteObject\} \subseteq \rho(o_{ExternalSource})$
- $\{open, close, createTransaction\} \subseteq \rho(o_{ExternalSource})$
- $ret(\alpha(o_{QueryAbleClass})(open)) = o_{Object}$
- $sig(\alpha(o_{QueryAbleClass})(open)) = ()$.
- $ret(\alpha(o_{QueryAbleClass})(close)) = o_{Object}$
- $sig(\alpha(o_{QueryAbleClass})(select)) = ()$.
- $ret(\alpha(o_{QueryAbleClass})(createTransaction)) = o_{ExternalTransaction}$.
- $sig(\alpha(o_{QueryAbleClass})(createTransaction)) = ()$.
- $ret(\alpha(o_{QueryAbleClass})(query)) = o_{qSetOfObject}$
- $sig(\alpha(o_{QueryAbleClass})(query)) = (o_{Query})$.
- $ret(\alpha(o_{QueryAbleClass})(getObject)) = o_{Tuple}$
- $sig(\alpha(o_{QueryAbleClass})(getObject)) = (o_{Query})$.
- $ret(\alpha(o_{QueryAbleClass})(updateObjectField)) = o_{Object}$
- $sig(\alpha(o_{QueryAbleClass})(updateObjectField)) = (o_{Query}, o_{String}, o_{Object})$.
- $ret(\alpha(o_{QueryAbleClass})(newObject)) = o_{Object}$
- $sig(\alpha(o_{QueryAbleClass})(newObject)) = ()$.
- $ret(\alpha(o_{QueryAbleClass})(deleteObject)) = o_{Object}$
- $sig(\alpha(o_{QueryAbleClass})(deleteObject)) = (o_{Query})$.

Nous appellerons transaction externe tout objet $o \in \widetilde{\pi}(o_{ExternalTransaction})$.

Notons que $o_{ExternalTransaction}$ hérite de $o_{Transaction}$. Dans notre modèle, toute transaction externe est en effet considérée comme sous-transaction d'une transaction de haut-niveau. De manière plus précise, lorsqu'une transaction souhaite accéder à une source de données externe pour la première fois par le biais d'un relai, une transaction externe correspondante est démarrée et associée à la racine de la hiérarchie transactionnelle dont fait partie la transaction considérée. La transaction externe sera alors validée ou annulée lors de la validation ou l'annulation de la transaction de haut-niveau à laquelle elle est associée. Notons que si les objets persistants accédés dans ce cadre sont associé à un comportement transactionnel, il faut alors que la validation ou l'annulation de ce comportement intervienne avant la validation de la transaction externe. Pour ce faire, il suffit que les relais, qui représentent les objets persistants, soient associés à des priorités de validation, d'annulation et de préparation plus fortes que celles associées aux transactions externes. Notons que ceci implique que les algorithmes de validation, de préparation ou d'annulation définis dans les méthodes de $o_{transaction}$ n'ont pas à être modifiés pour prendre en compte les objets persistants.

Nous appellerons source de données externe tout objet $o \in \widetilde{\pi}(o_{ExternalSource})$. Nous supposerons que les spécificités de chaque source peuvent être spécifiées au sein des méthodes associées aux messages de $\rho(o_{ExternalSource})$ définis précédemment. En particulier :

- la méthode associée au message *query* doit traduire la requête passée en argument en requête compréhensible par la source de données locale et renvoyer le résultat correspondant.

- la méthode associée au message *getObject* retourne le n-uplet correspondant à l'entitée persistante identifiée par la requête passée en argument.
- la méthode associée au message *updateObjectField* doit mettre à jour la donnée stockée au sein de la base correspondant au champ, nommé par la chaîne de caractère passée en argument, de l'entité de la base désignée par la requête passée en argument, et ce en fonction de la nouvelle valeur passée en argument.
- la méthode associée au message *deleteObject* doit supprimer la donnée désignée par la requête passée en argument.

3.3.2.2 Formalisation des relais

classes de relais et relais Soit $o_{ProxyClass}$ et $o_{ProxyObject}$ deux éléments de O tels que :

- $o_{ProxyObject} = root(o_{ProxyClass})$;
- $o_{ProxyClass} \prec o_{TransactionAbleClass}$;
- $o_{ProxyClass} \prec o_{QueryAbleClass}$;
- $o_{ProxyClass} \dashv o_{Class}$;
- $o_{ProxyObject} \prec o_{TransactionAbleObject}$;
- $o_{ProxyObject} \prec o_{QueryAbleObject}$;
- $o_{ProxyObject} \prec o_{Transaction}$;
- $o_{ProxyObject} \dashv o_{ProxyClass}$.

Nous appellerons classe de relai tout objet c tel que $c \prec o_{ProxyObject}$ et $c \dashv o_{ProxyClass}$.

Nous nommerons $\mathcal{C}_r$, l'ensemble des classe de relais.

Nous appellerons relai tout objet o instance d'une classe de relai.

Nous nommerons $\mathcal{O}_r$, l'ensemble des relais.

Fonctions d'identifications Nous appelons fonctions d'identification, nommées respectivement *base*, *name* et *id*, les fonctions définies de la manière suivantes :

- *base* : cette fonction, de $\mathcal{C}_r$ dans $\widetilde{\pi}(o_{ExternalSource})$, associe à chaque classe de relai la base au sein de laquelle sont stockées les données représentées par cette classe.
- *name* : cette fonction de $\mathcal{C}_r$ dans $\mathcal{S}$, associe à chaque classe de relai le nom de l'entité structurelle que représente cette classe.
- id_c : cette fonction de $\mathcal{C}_r$ dans $\mathcal{S}$, associe à chaque classe de relai l'expression sous la forme de chaîne de caractères permettant d'identifier de manière unique au sein de la base chaque donnée représentée par un relai instance de cette classe. Par exemple, si la base sous-jacente est une base de données relationnelles, cette expression peut-être le nom du champ contenant la clé primaire de la table considérée.

Fonctions de contenu Les relais sont la représentation en mémoire des données stockées au sein de bases de données externes. S'il est nécessaire de disposer d'autant de relais que de résultats des requêtes effectuées sur une base pour permettre la référenciation de ces données, il n'est pas nécessaire d'avoir ces données en permanence en mémoire. Dans notre système, nous considérons qu'un relai peut contenir soit la référence à une donnée persistante, soit la donnée elle-même. Bien évidemment, lors de l'accès à un relai, si celui-ci ne contient que l'identification

de la donnée, celle-ci doit être chargée en mémoire pour permettre son exploitation. Pour éviter de surcharger la mémoire, il peut être nécessaire lors du chargement d'une donnée d'en décharger une autre. Nous permettons à l'utilisateur de notre système de préciser les données pouvant être déchargées et/ou le nombre de données pouvant être présentes en mémoire par classe de relais.

Pour formaliser cela, nous définissons les fonctions suivantes, nommées respectivement id_o, $data$, max, min et $isUnloadable$.

- id_o : cette fonction, de $\mathcal{O}_r$ dans O, associe à chaque relai o la valeur de son identifiant au sein de la base correspondant à l'expression $id_c(\gamma(o))$.
- $data$: cette fonction, de $\mathcal{O}_r$ dans O, associe à chaque relai o la donnée lui correspondant. Notons que le résultat de cette fonction peut-être o_{Void}[6]si la donnée n'est pas chargée en mémoire.
- max : cette fonction, de $\mathcal{C}_r$ dans $\widetilde{\pi}(o_{int})$, associe à chaque classe de relais le nombre maximal de relais instance de cette classe pouvant posséder une donnée chargée.
- min : cette fonction, de $\mathcal{C}_r$ dans $\widetilde{\pi}(o_{int})$, associe à chaque classe de relais le nombre minimal de relais, instances de cette classe, devant être déchargés.
- $isUnloadable$: cette fonction de $\mathcal{O}_r$ dans $\widetilde{\pi}(o_{Boolean})$, permet de déterminer si la donnée d'un relai peut-être déchargée ou non. Notons que la fonction min est prioritaire sur cette fonction. Si le maximum est atteint et s'il n'est pas possible de trouver suffisament de relais dont les données peuvent être déchargées, les données des relais dont cette fonction renvoie o_{false} seront tout de même déchargées jusqu'à atteindre le nombre minimal de déchargement.

De plus, nous souhaitons permettre la mise à jour des données persistantes ainsi que leur supression ou leur création. Du point de vue formel, nous traduisons cela par trois nouvelles fonctions nommées respectivement $memObjects$, $newObjects$ et $deletedObjects$ et par un message pouvant être envoyé aux relais, nommé $save$:

- $memObjects$: cette fonction de $\mathcal{C}_r$ dans $P^{fin}(\mathcal{O}_r)$, associe à chaque classe de relais l'ensemble de ses instances dont les données persistantes sont chargées en mémoire et donc modifiables.
- $newObjects$: cette fonction de $\mathcal{C}_r$ dans $P^{fin}(O_r)$, associe à chaque classe de relais l'ensemble de ses instances créées par l'application et dont les données ne sont pas encore persistantes.
- $deletedObjects$: cette fonction de $\mathcal{C}_r$ dans $P^{fin}(\mathcal{O}_r)$, associe à chaque classe de relais l'ensemble de ses instances supprimées par l'application et dont les données persistantes doivent être supprimées de la base sous-jacente.
- $save$: ce message, élément de $\rho(o_{ProxyObject})$ est tel que $ret(\alpha(o_{ProxyObject})(save)) = o_{Void}$ et $sig(\alpha(o_{ProxyObject})(save)) = ()$. L'exécution de la méthode correspondante met à jour les données modifiées contenues dans le relai à l'aide de messages $updateObjectField$ envoyés à la base associée au relai considéré. Notons que lorsqu'il s'agit de mettre à jour un champ atomique, il suffit de passer en argument la nouvelle valeur. Lorqu'il s'agit d'une référence sur d'autres données (par exemple dans le cas d'une base de données à objets), il faut alors renvoyer l'expression permettant de calculer la valeur du champ référence au sein de la base.

Algorithme de traitement des requêtes Cet algorithme correspond d'une part à l'exécution de la méthode induite par l'envoi d'un message $\lambda'(c, select, (r))$ avec c la classe de relais

[6]Pour mémoire, o_{Void} référence un objet représentant l'absence de valeur.

sur laquelle est émise la requête r et, d'autre part, à l'exploitation de l'ensemble résultant. Il suit les règles décrites ci-dessous :

- $\lambda'(c, select, (r)) = e$ défini de la façon suivante :
  - soit $e' = \lambda'(base(c), select, (r'))$ avec $r' \in \widetilde{\pi}(o_{Query})$ tel que $where(r') = where(r)$ et $proj(r') = id_c(c)$,
  - soit $id_1 = \{o \in resultElements(e') \mid \exists o' \in \widetilde{\pi}(c) \mid id_o(o') = o\}$,
  - soit $id_2 = resultElements(e') \setminus id_1$, pour chaque élément de id_2, il faut créer le relais correspondant,
  - soit $id_3 = id_1 \setminus \{o \in id_1 \mid id_o^{-1}(o) \in memObjects(c) \cup deletedObjects(c)\}$
  - soit $r_1 = \{o \in memObjects(c) \cup newObjects(c) \mid \lambda'(r, applyWhere, (o)) = o_{true}\}$
  - soit $r_2 = \{o \mid id(o) \in id_2 \wedge \lambda'(r, applyWhere, (o)) = o_{true}\}$ (Notons que nous devons exécuter à nouveau la requête en mémoire uniquement dans le cas où la traduction de la requête en requête émise sur la base n'est pas complète)
  - soit $r_3 = \{o \mid id(o) \in id_3 \wedge \lambda'(r, applyWhere, (o)) = o_{true}\}$ (même remarque que précédemment)
  - $query(e) = r, elements(e) = r_1 \cup r_2 \cup r_3$.
- De la même manière que lors de l'accès au résultat d'une requête effectuée en environnement volatile, $\lambda'(e, getElement, ()) = o$ tel que $\exists o' \in elements(e)$, $\lambda'(query(e), applyProj, (o')) = o$.

Accès aux champs d'un relai Comme précédemment, nous supposons que l'accès aux champs d'un relai ne peut-être effectué que par le biais d'un accesseur.

Lorsqu'un relai o est accédé, deux cas sont possibles :

- Si les données persistantes n'ont pas été chargées, c'est-à-dire si $data(o) = o_{Void}$, avant de retourner ou de modifier la valeur concernée il faut alors d'une part charger les données correspondantes, et d'autre part vérifier que l'on ne dépasse pas la limite de chargement autorisé, c'est-à-dire il faut vérifier que le nombre d'éléments de $memObjects(\gamma(o))$ soit inférieur à $max(\gamma(o))$. Dans le cas contraire, il faut alors supprimer au moins $min(\gamma(o))$ données associés aux objets instances de $\gamma(o)$ en commencant par supprimer celles autorisées par la fonction $isUnloadable$.
- Sinon, il suffit de retourner ou de modifier la valeur concernée.

De manière plus formelle, pour tout $o \in \mathcal{O}_r$ si le message $\lambda'(o, s, a)$ est envoyé tel que $\alpha(\gamma(o))(s) \in \widetilde{\pi}(o_{Accessor})$ alors l'algorithme exécuté est le suivant.

- Si $data(o) = o_{Void}$

  1. Construire la requête $r \in \widetilde{\pi}(o_{Query})$ telle que
     - $where(r)$ contienne une chaîne de caractères identifiant les données recherchées construite trivialement à partir de $id_c(\gamma(o))$ et de $id_o(o)$.
     - $proj(r)$ contienne les noms des slots de type atomique et les expressions permettant de construire les identifiants référencant les eventuelles données liées par réfèrerence aux données chargées. Supposons en effet que nous souhaitions accéder à une base de données objet. Lorsque l'on charge un objet de la base, pour ne pas avoir à charger l'ensemble des objets qui lui sont liés, ce qui peut à l'extrême impliquer le chargement de la base en entier, il faut non pas demander à la base de charger l'objet entier en

mémoire mais uniquement les données atomiques qui le composent et les références aux autres objets qui lui sont liés.

2. Le n-uplet $t = \lambda'(base(\gamma(o)), getObject, (r))$, dans lequel les références aux données liées sont remplacées par les relais correspondants est alors associé à l'objet courant, c'est-à-dire que désormais, $data(o) = t$
3. Puis, si $memObjects(\gamma(o)) \geq max(\gamma(o))$ alors considérons $e = \{o \in memObjects(\gamma(o)) \mid isUnloadable(o) = o_{True}\}$. Si $card(e) < min(\gamma(o))$ il faut rajouter à e autant d'éléments de $memObjects(\gamma(o))$ qu'il est nécessaire pour que le nombre d'élémnts de e soit au moins égal à $min(\gamma(o))$. Puis, il faut sauvegarder (message *save*) et supprimer de la mémoire les données de tous les relais éléments de e, c'est-à-dire, il faut qu'à la suite de l'opération $\forall o \in e$, $data(o) = o_{Void}$. Enfin, il faut supprimer tous ces relais de l'ensemble résultat de $memObjects(\gamma(o))$, c'est-à-dire, il faut qu'à la suite de l'opération $memObjects(\gamma(o)) = memObjects(\gamma(o)) \setminus e$.
4. Pour terminer, il suffit de rajouter o à $memObjects(\gamma(o))$ et de retourner ou modifier la donnée correspondant à l'accesseur.

- Sinon, il suffit de retourner ou modifier la donnée correspondant à l'accesseur.

3.3.2.3 Relais et transactions

Les relais, tels que nous les avons définis peuvent être considérés comme des objets transactionnels. Ils peuvent donc être associés de la même manière à des comportements transactionnels. Comme, notre système repose avant tout sur l'envoi de message, l'association d'un relai à un comportement transactionnel est identique à l'association d'un objet transactionnel à un comportement transactionnel. Sans aucune modification des algorithmes liés au traitement transactionnel, les relais peuvent être gérés par notre système transactionnel.

De plus, notre système transactionnel permet un accès transparent aux sources de données externes sans nécessiter la gestion explicite par l'utilisateur de transactions sur ces sources de données. Pour mémoire, lorsqu'une transaction de haut-niveau valide ou annule, par le biais de l'algorithme de validation en 2 phases et des priorités données à chaque entité du système, les relais mis en œuvre dans cette transaction sont tout d'abord traités comme des objets transactionnels standard puis la transaction externe leur correspondant est validée ou annulée.

Lorsqu'une transaction de haut-niveau est associée à plusieurs transactions externes, on se retrouve alors face aux problèmes présentés dans le chapitre I concernant l'atomicité et l'isolation des transactions ainsi que les problèmes de verrous mortels liés à un environnement distribué hétérogène. Nous avons vu dans le premier chapitre que ces problèmes n'avaient pas de solutions satisfaisantes, mais qu'il était possible de limiter leurs effets. Nous décrivons ci-dessous l'approche que nous avons adopté :

- En ce qui concerne l'atomicité, nous permettons de définir des opérations de préparation pour les transactions externes. Les systèmes exportant l'opération de préparation peuvent donc être gérés de manière optimale. Pour les autres, il est par exemple possible, pour limiter le risque d'incohérence de leur associer une priorité plus faible pour qu'elle ne soient traitées, lors de la validation, que lorsque toutes les autres transactions ont déjà été traitées.

- En ce qui concerne l'isolation, elle peut-être traitée par notre système transactionnel en mémoire. Il suffit en effet d'associer à chaque relai un comportement transactionnel garantissant l'isolation des transactions y accédant.
- En ce qui concerne les verrous mortels, ceux étant liés aux algorithmes d'isolation mis en œuvre par notre système peuvent être détectés. Pour les autres, induits par une attente lors de la lecture ou l'écriture d'une donnée persistante dans la base, nous adoptons une approche préventive en posant un délai de garde à l'exécution d'un chargement de données (dans le cadre d'un accesseur) ou lors de la sauvegarde de données (message save). Il peut-être intéressant (voire utile) de permettre la définition de ce délai de garde pour chaque classe de relai par l'utilisateur. Nous ne formalisons pas cette technique qui relève du domaine de l'implantation.

Notons que pour permettre la validation ou l'annulation de la transaction externe associée au relai, il faut que les relais qui lui correspondent soient dans un état cohérent par rapport à leur propriété de persistance. En particulier, il est nécessaire d'effectuer les tâches suivantes avant de valider une transaction externe.

- sauvegarder les modifications effectuées sur les relais dont les données sont encore en mémoire,
- supprimer de la base les données marquées comme tel.
- ajouter dans la base les données créées lors de la transaction

Ces différentes tâches sont liées à la classe des relais considérés, et plus particulierement à la gestion de l'extension de cette classe, ou tout du moins à la gestion du sous-ensemble de l'extension de cette classe dont les données sont en mémoire. Pour effectuer ces tâches de manière transparente, il est intéressant de considérer les classes de relais comme des objets transactionnels particuliers dont le comportement de validation corresponde à effectuer ces tâches et le comportement d'annulation à supprimer des ensembles concernés les objets créés ou supprimés selon les cas. C'est pourquoi, $o_{ProxyClass}$ hérite de $o_{Transaction}$. Notons que ces comportements transactionnels doivent être appelés avant ceux de la transaction externe concernée mais après ceux des relais. Si l'on considère que les relais ont une priorité haute comme les autres objets transactionnels et que les transactions externes ont une priorité basse, les classes de relais doivent avoir une priorité moyenne.

3.4 Vues

Pour terminer ce chapitre, attachons nous à introduire les vues au sein de notre système. Pour ce faire, nous présentons ici une extension du métamodèle de notre système pour permettre l'exploitation des concepts de classe virtuelle et de mapping.

3.4.1 Rappels et discussion

3.4.1.1 Rappels

Pour mémoire, nous présentons ci-dessous les différents concepts que nous souhaitons mettre en œuvre dans le cadre de l'intégration des vues au sein de notre système :

- une classe virtuelle est l'équivalent virtuel d'une classe classique. Ces instances sont toutes structurées suivant le même type et peuvent toutes faire l'objet des mêmes messages.
- Un mapping est la description des correspondances entre la structure et le comportement de la classe virtuelle à laquelle il est associé et la structure et le comportement d'une ou plusieurs classes de base. Un mapping est, au sein de notre système, représenté par une classe.
- Chaque instance d'un mapping, appelé objet virtuel, est structurée suivant le type de la classe virtuelle auquel ce mapping est associé, est relative à une instance de chaque classe de base du mapping, dit objet de base, et est valuée suivant le résultat d'un calcul fonction de ces instances de bases.
- Chaque instance d'un mapping peut recevoir les messages définis sur la classe virtuelle auquel ce mapping est associé. Chacun de ces messages résulte en l'exécution de la première méthode trouvée qui vérifie l'une au moins des propriétés suivantes :

  1. une méthode virtuelle définie au sein du mapping dont la valeur de retour est le résultat d'un calcul fonction du résultat de l'exécution de methodes sur chaque instance de base ;
  2. une méthode définie au sein du mapping ;
  3. une méthode définie au sein de la classe virtuelle.

- L'extension d'une classe virtuelle est constituée de l'union de l'extension de chacun des mappings auxquels elle est associée.

3.4.1.2 A propos de la matérialisation

L'intégration des vues au sein de notre système peut être réalisée suivant deux approches. Il est en effet possible de considérer que les vues doivent être matérialisées ou non. En d'autres termes, avant d'aller plus loin, il est nécessaire de déterminer si les objets virtuels doivent contenir les données résultats des calculs permettant leur exploitation, ou s'il faut effectuer ces calculs lors de chaque accès aux objets virtuels.

- La première approche consiste à considérer les classes virtuelles comme des relais plus complexes que ceux utilisés lors de l'accès aux données persistantes.
  - L'avantage de cette approche est, lors de la lecture d'une donnée d'un objet virtuel, de n'avoir à effectuer de calcul que si les données utilisées dans celui-ci n'ont pas été modifiées depuis le calcul précédent. Si c'est le cas, il suffit en effet de lire la sauvegarde du calcul précédent.
  - En contrepartie, les données stockées au sein d'un objet virtuel sont redondantes avec celles contenues dans les objets de base. En conséquence, si un objet virtuel est basé sur un ou plusieurs objets persistants, des données stockées dans la ou les bases de données correspondantes peuvent être représentées deux fois en mémoire. Nous avons vu que les relais utilisent un système de cache pour limiter le nombre de données persistantes présentes en mémoire. Pour garantir cette propriété lors de la gestion des vues, il est nécessaire de mettre en œuvre au niveau des mappings ou des classes virtuelles un système de cache similaire à celui utilisé pour les relais. Dans ce cas la matérialisation des objets virtuels ainsi que ses avantages n'est plus que temporaire.
  - De plus, la redondance des données entre objets virtuels et objets de base nécessite la mise en œuvre d'une part d'algorithmes de mises à jour des objets de base vers les objets

virtuels et, d'autre part, des algorithmes de gestion transactionnelle. En effet, si un objet est modifié alors qu'un objet virtuel basé sur cet objet a matérialisé le résultat des calculs, il est nécessaire de l'informer de la modification pour qu'il puisse lors de son prochain accès recalculer les données matérialisées. D'une part, cela implique que chaque objet puisse connaître les objets virtuels qui sont construits à partir de lui. D'autre part, cela induit un surcoût lors de chaque modification de l'objet de base, celui-ci devant notifier les objets virtuels concernés. Enfin, si l'on permet de mettre à jour les objets virtuels et si ceux-ci sont matérialisés, il se pose alors des problèmes de gestion transactionnelle sur ces objets. Il faut en effet faire en sorte que les algorithmes de gestion transactionnelle utilisés pour gérer les données matérialisées soient cohérents avec la gestion transactionnelle des données utilisées lors du calcul dont elles sont le résultat. Par exemple, considérons un objet virtuel o basé sur un seul objet o_1. Supposons que o_1 soit géré par un algorithme de contrôle de concurrence par certification tel que celui présenté dans le chapitre I. Dans ce cas, il est possible que deux transactions concurrentes sur o_1 le modifient, ce qui résulte en de nouvelles versions de o_1 associées à chaque transaction. Chaque version doit ensuite notifier o qu'elles ont été modifiées. Cet objet virtuel doit donc tout d'abord faire la différence entre deux modifications consécutives et deux modifications effectuées par deux transactions distinctes. De plus, lors de l'accès par chacune des transactions aux données qu'il contient, il doit calculer puis matérialiser la version correspondante aux données accédées. Dans ce cas particulier, on voit qu'un objet virtuel doit non seulement être géré de manière transactionnelle mais que cette gestion doit être cohérente avec celle des objets sur lesquels il est basé. Or si on n'impose aucune contrainte sur les objets de base, ceux-ci peuvent être gérés de différentes manières transactionnelles qui ne sont pas nécessairement compatibles entre elles. Cela implique qu'il peut-être difficile voire impossible de déterminer le comportement transactionnel à adopter pour l'objet virtuel.

- La seconde approche consiste à ne considérer les classes virtuelles que comme des entités fonctionnelles dont les instances ne contiennent aucune donnée. Dans ce cas, il est évident que les défauts de l'approche précédente n'existent plus. Néanmoins, il est cette fois nécessaire de calculer les données des objets virtuels à chaque accès, que les objets de base aient été modifiés ou non, ce qui grève les performances du système.

Si l'on ne raisonne qu'aux niveaux des performances du système, la première approche est plus intéressante que la seconde si la somme des temps d'exécution liés à la matérialisation des données, à la notification de mise à jour et au contrôle transactionnel de chaque objet virtuel est inférieur au temps de calcul de celui-ci. De manière plus précise, soit c le temps de calcul d'un objet virtuel, soit c' le temps de matérialisation d'un objet virtuel, soit c_t le temps de gestion transactionnelle lors d'un accès à un objet virtuel et soit k le nombre d'accès à un objet transactionnel alors la première approche est intéressante si $c + c' + k \times c_t < k \times c$, ceci en supposant que les objets virtuels ne sont pas gérés à l'aide d'un cache et que le temps de notification de mise à jour est négligeable, ce qui est favorable à la première approche. En supposant de plus que $k > 1$, la première approche est donc intéressante si $c > \frac{c'}{k-1} + \frac{k}{k-1} c_t$. Lorque k est grand, la première approche est intéressante si $c > c_t$. Comme nous avons pu le voir précédemment, la gestion transactionnelle d'un objet est très couteuse. Pour que la première approche soit intéressante, il faut donc que, dans le meilleur des cas (c'est-à-dire lorsqu'il y a peu de mises à jour des objets de base et lorsqu'on ne gère pas de cache), le calcul des objets virtuels soit très complexe.

De plus, comme nous l'avons précisé ci-dessus, il est difficile de déterminer le type de gestion transactionnelle à utiliser pour les objets virtuels. En conséquence, nous adoptons au sein de notre système la seconde approche.

3.4.1.3 Structures et Comportements

Comme nous venons de le voir, nous ne souhaitons pas matérialiser les vues. En conséquence, la structure d'un objet virtuel est réduite au minimum permettant d'identifier l'objet virtuel. L'identification d'un objet virtuel doit en fait jouer un double rôle. D'une part, un objet virtuel doit pouvoir être identifié par le système comme n'importe que autre objet du système. En ce sens, l'identifiant d'un objet virtuel doit être élément de O, ensemble des identifiants d'objets. D'autre part, un objet virtuel doit aussi pouvoir être identifié en fonction des objets à partir desquels il est calculé. En particulier, si l'un de ces objets devait être supprimé ou ne plus vérifier les règles décrites dans le mapping dont l'objet virtuel est instance, celui-ci devrait aussi être supprimé. A l'inverse, un objet virtuel ne peut être créé que si ses objets de bases existent déjà où sont créés en même temps que lui. L'identification des objets virtuels est donc sujet à discussion et modélisation (voir par exemple [Bellahsene97] ou [Damodoran-Kamal et al.94]). Nous avons choisi de séparer l'identifiant pour le système de l'identifiant associé aux objets de base en définissant ce dernier comme l'unique donnée contenue dans l'objet virtuel considéré. Cela permet de garder une certaine homogénéité entre les objets et les objets virtuels et de ne pas avoir à modifier le système d'identification.

Pour en revenir à la structure virtuelle des objets virtuels, celle-ci doit, comme nous l'avons précédemment précisé, pouvoir être mise en correspondances avec les structures des objets de base. Comme nous avons choisi de ne pas matérialiser des objets virtuels, leur structure, qui n'est que virtuelle, est décrite dans notre système par le biais d'accesseurs définis au sein des classes virtuelles. Dans ce cadre, les correspondances entre les structures virtuelles et leurs structures de base sont décrites entre les accesseurs définis dans la classe virtuelle et les méthodes definies dans les classes de base, et ce de manière identique aux correspondances entre les méthodes virtuelles et les méthodes de base. Notons que ces correspondances peuvent aussi bien porter sur des accesseurs en lecture, que sur des accesseurs en écriture. Dans ce dernier cas, cela permet de mettre à jour les objets virtuels par le biais de la mise à jour ou de l'exécution de méthode sur ses objets de base.

Pour permettre la mise en correspondance de plusieurs classes de base dans le but de former une classe virtuelle, nous ajoutons à la description des mappings la possibilité de définir deux contraintes. La première, décrite sous la forme d'une chaîne de caractères contient une expression conditionnelle qui doit permettre au système, lorsqu'il souhaite exécuter une requête sur une classe virtuelle, de préciser et donc d'optimiser les requêtes envoyées aux classes de base pour former le résultat. La seconde, décrite sous la forme d'une méthode dont le type de retour est un booléen permet de définir une contrainte nécessitant l'exécution d'un algorithme complexe. En contrepartie, cette contrainte ne peut être utilisée lors de l'exécution d'une requête.

Enfin, pour permettre la création de nouveaux objets virtuels ou leur destruction, un mapping doit contenir deux méthodes décrivant l'algorithme permettant de construire et/ou détruire les objets de base correspondants.

3.4.2 Formalisation

3.4.2.1 Classes virtuelles et mappings

Nous introduisons ci-dessous les concepts de classe virtuelle et de mapping au sein de notre formalisme.

Classe virtuelle Soient $o_{VirtualClass}$ et $o_{VirtualObject}$ deux éléments de O tels que :

- $root(o_{VirtualClass}) = o_{VirtualObject}$
- $o_{VirtualClass} \prec o_{QueryAbleClass}$ et $o_{VirtualClass} \dashv o_{Class}$.
- $o_{VirtualObject} \prec o_{QueryAbleObject}$ et $o_{VirtualObject} \dashv o_{VirtualClass}$.

Nous appellerons $\mathcal{C}_v$ l'ensemble des classes virtuelles, c'est-à-dire l'ensemble des objets o tels que $o \in \widetilde{\pi}(o_{VirtualClass})$.

Mapping Soient $o_{MappingClass}$ et $o_{MappingObject}$ deux éléments de O tels que :

$root(o_{MappingClass}) = o_{MappingObject}$

$o_{MappingClass} \prec o_{QueryAbleClass}$ et $o_{MappingClass} \dashv o_{Class}$.

$o_{MappingObject} \prec o_{QueryAbleObject}$ et $o_{MappingObject} \dashv o_{MappingClass}$.

Nous appellerons $\mathcal{C}_m$ l'ensemble des mappings, c'est-à-dire l'ensemble des objets o tels que $o \in \widetilde{\pi}(o_{MappingClass})$.

Nous appellerons $\mathcal{O}_v$ l'ensemble des objets virtuels, c'est-à-dire l'ensemble des objets o tels que $o \in \widetilde{\pi}(o_{MappingObject})$.

Fonction de structuration et fonction de mapping Nous appellerons fonction de structuration, nommée *struct*, une fonction de $\mathcal{C}_m$ dans $\mathcal{C}_v$ associant à chaque mapping une classe virtuelle.

Tout mapping $c \in \mathcal{C}_m$ doit être tel que $\rho(c) \supseteq \rho(struct(c))$.

Nous appellerons fonction de mapping, nommée *map*, une fonction de $\mathcal{C}_v$ dans $P^{fin}(\mathcal{C}_m)$ associant à chaque classe virtuelle l'ensemble des mappings actifs sur cette classe. Un mapping est dit actif s'il est pris en compte lors du calcul de l'extension virtuelle d'une classe virtuelle. Notons que la fonction *map* n'est pas « l'inverse », au sens mathématique du terme, de la fonction *struct*.

Nous appellerons instance virtuelle d'une classe virtuelle, l'instance d'un mapping actif associée à cette classe.

Nous appellerons extension virtuelle d'une classe virtuelle c, notée $\widetilde{\pi}_v(c)$, l'ensemble des instances virtuelles de cette classe, soit l'union des extensions des mappings actifs associés à c, et nous imposons que $\forall c \in \widetilde{\pi}(o_{VirtualClass}), \widetilde{\pi}(c) = \widetilde{\pi}_v(c)$.

Correspondances Soit $o_{Mapping}$ élément de O tel que

- $o_{Mapping} \prec o_{Object}$ et $o_{Mapping} \dashv o_{Class}$.
- $\{apply\} \in \rho(o_{Mapping})$

Nous appellerons Ξ l'ensemble des correspondances, c'est-à-dire l'ensemble des objets $o \in \widetilde{\pi}(o_{Mapping})$.

Soit $o_{SimpleMapping}$ élément de O vérifiant les deux propriété suivantes : $o_{SimpleMapping} \prec o_{Mapping}$ et $o_{SimpleMapping} \dashv o_{Class}$.

Soit $o_{ComplexMapping}$ élément de O vérifiant les deux propriété suivantes : $o_{ComplexMapping} \prec o_{Mapping}$ et $o_{ComplexMapping} \dashv o_{Class}$.

Les objets instances de $o_{SimpleMapping}$ représentent des correspondances simples entre un message envoyé à un objet virtuel et le message à envoyer à l'un des objets de base.

Les objets instances de $o_{ComplexMapping}$ représentent des correspondances complexes entre un message envoyé à un objet et les messages à envoyer à plusieurs objets de base ainsi que la méthode de combinaison des résultats de ces messages.

Nous supposerons que le résultat de l'envoi d'un message *apply* sur un objet correspondance quel qu'il soit est le résultat du message associé dans le cas d'une correspondance simple ou de la combinaison des résultats des messages associés dans le cas d'une correspondance complexe.

Fonction de correspondances et fonction de contraintes Nous appellerons fonction de correspondances, nommée $corr$, une fonction, de $\mathcal{C}_m \times \mathcal{S}$ dans Ξ associant à un mapping m et à un message élément de $\rho(struct(m))$ une correspondance.

Nous appellerons fonction de contrainte simple, nommée $cont_s$, une fonction de $\mathcal{C}_m$ dans $\mathcal{S}$ associant à chaque mapping une expression sous forme de chaîne de caractères contraignant les objets des classes de base pouvant faire partie d'un objet virtuel.

Nous appellerons fonction de contrainte complexe, nommée $cont_c$, une fonction de $\mathcal{C}_m$ dans $\mathcal{B}$ associant à chaque mapping une méthode contraignant les objets des classes de base pouvant faire partie d'un objet virtuel.

Fonction de Base Nous appellerons fonction de base de mapping, nommée $base_m$, une fonction de $\mathcal{C}_m$ dans $P^{fin}(\widetilde{\pi}(o_{QueryAbleClass}))$ associant à chaque mapping l'ensemble des classes sur lequel il est basé. Notons qu'une classe de base doit nécessairement être instance de $o_{QueryAbleClass}$. En effet, il doit pouvoir être possible d'effectuer des requêtes portant sur les classes virtuelles. Comme nous le verrons dans la suite, l'envoi d'une requête sur une classe virtuelle peut impliquer l'envoi de requêtes sur les classes de base des mappings qui lui sont associés. En conséquence, les classes de base doivent pouvoir traiter le message *select*.

Nous appellerons fonction de base virtuelle, nommée $base_v$, une fonction de $\mathcal{O}_v$ dans $L^{fin}(O)$ associant à chaque objet virtuel la liste des objets sur lesquel il est basé telle que $\forall o \in \mathcal{C}_m, \forall o' \in base_v(o), \exists c \in base_m(\gamma(o)) \mid c = \gamma(o')$ et telle que $\forall o \in \mathcal{C}_m$, $base_v(o)$ vérifie les contraintes relatives à $cont_s(\gamma(o))$ et $cont_c(\gamma(o))$. Notons que cette fonction peut-être considérée comme la fonction d'identification des objets virtuels à partir de leurs objets de base.

Envoi de message L'envoi d'un message $\lambda'(o, s, a)$ tel que $o \in \mathcal{O}_v$ correspond à envoyer le message *apply* sur la correspondance associée au message. En d'autres termes, on doit avoir : $\lambda'(o, s, a) = \lambda'(corr(\gamma(o), s), apply, (o, a))$.

Vues et héritage D'après les règles de compatibilité définies dans le chapitre précédent, il est possible qu'une classe virtuelle ou qu'un mapping soit sous-classe d'une classe non virtuelle. D'après les mêmes règles, il est à l'inverse impossible qu'une classe non virtuelle soit sous-classe d'une classe virtuelle ou d'un mapping.

Intéressons nous maintenant aux problèmes de compatibilité entre classes virtuelles et mappings lorsque des liens d'héritage sont mis en jeu. Nous avons vu que l'extension virtuelle d'une classe virtuelle est égale à l'union des extensions des mappings actifs qui lui sont associés. Soit $c \in \widetilde{\pi}(o_{VirtualClass})$, on a donc $\widetilde{\pi}_v(c) = \bigcup_{m \in map(c)} \widetilde{\pi}(m)$. Par ailleurs, supposons d'une part que l'extension virtuelle d'une classe virtuelle soit, par analogie avec l'extension d'une classe, égale à l'union des extensions virtuelles de ses sous-classes et de l'ensemble de ces instances virtuelles propres, et d'autre part, que cet ensemble, noté π_v, soit égal à $\bigcup_{m \in map(c)} \pi(m)$. Soit $c \in \widetilde{\pi}(o_{VirtualClass})$, on a donc $\widetilde{\pi}_v(c) = (\bigcup_{c' \prec c} \pi_v(c'))$. En conséquence, il vient $\bigcup_{m \in map(c)} \widetilde{\pi}(m) = (\bigcup_{c' \prec c} \pi_v(c'))$, soit $\bigcup_{m \in map(c)} (\bigcup_{m' \prec m} \pi(m')) = \bigcup_{c' \prec c} (\bigcup_{m'' \in map(c')} \pi(m''))$. Comme $\forall c_1, c_2 \in \mathcal{M}_o, c_1 \neq c_2 \Rightarrow \pi(c_1) \cap \pi(c_2) = \emptyset$, pour toute c', sous-classe de c, et pour tout m'' mapping actif de c'., d'après l'égalité précédente, il existe un mapping m', sous-classe d'un mapping actif m associé à c tel que $m'' = m'$.

En conséquence, si l'on souhaite que la sémantique du lien d'héritage soit vérifiée pour les classes virtuelles, il est nécessaire que $\forall c, c' \in \widetilde{\pi}(o_{VirtualClass})$, si $c \prec c'$ alors $\forall m' \in map(c')$ il existe $m \in map(c)$ tel que $m' \prec m$.

Notons que les contraintes d'héritage posées sur les classes virtuelles et sur les mappings, dérivent directement des propriétés générales de notre formalisme et non d'une simplification garantissant la cohérence des vues mais non fondée théoriquement telle que celles présentées en [Bellahsene97] ou [Souza dos Santos95].

3.4.2.2 Gestion des requêtes

Pour terminer ce chapitre, nous présentons les techniques de gestion de requêtes appliquées aux vues

Comme nous l'avons déjà précisé, l'extension virtuelle d'une classe virtuelle est l'union des extensions de chacun des mappings actifs associés à cette classe virtuelle. En conséquence, le résultat d'une requête concernant une classe virtuelle est l'union des résultats de cette même requête exécutée sur chacun des mappings. Notons que les classes virtuelles et les mappings peuvent faire l'objet de requêtes car elles sont instances de $o_{QueryAbleClass}$ et héritent de $o_{QueryAbleObject}$.

Comme précédemment, nous considérons qu'une requête est composée d'une expression booléenne formant la restriction de la requête et d'une expression formant la projection de la requête. Nous supposons de plus que le résultat d'une requête est instance de $o_{qSetOfObject}$. Le traitement des requêtes émises sur les classe virtuelles ne diffère donc du traitement des requêtes émises sur des classes standards ou sur des relais que par le traitement de leur clause de restriction.

Nous décrivons ci-dessous ce traitement dans le cas de classes virtuelles et dans le cas des mappings.

Classes virtuelles L'exécution d'une requête sur une classe virtuelle correspond, en premier lieu, à chercher pour chaque mapping actif, associé à la classe virtuelle, les objets virtuels vérifiant la restriction de la requête. Ensuite, l'ensemble de ces objets est retourné sous la forme d'une instance de $o_{qSetOfObject}$ associée à la requête. La projection n'est enfin exécutée que lors de l'accès aux éléments de cet ensemble.

De manière plus formelle, cet algorithme correspond d'une part à l'exécution de la méthode induite par l'envoi d'un message $\lambda'(c, select, (r))$ avec c la classe virtuelle sur laquelle est émise la requête r et d'autre part à l'exploitation de l'ensemble résultant. Il suit les règles décrites ci-dessous :

- $\lambda'(c, select, (r)) = e$ tel que :
  - pour tout $m \in map(c)$, soit $e'_m = \lambda'(m, select, (r))$,
  - $query(e) = r, elements(e) = \bigcup_{m \in map(c)} elements(e'_m)$.
- De la même manière que lors de l'accès au résultat d'une requête effectuée en environnement volatile ou persistant, $\lambda'(e, getElement, ()) = o$ tel que $\exists o' \in elements(e)$ vérifiant $\lambda'(query(e), applyProj, (o')) = o$.

Mappings en environnement volatile Tout d'abord considérons le cas où les mappings ne peuvent être associés à des classes de relais. En d'autres termes, considérons uniquement le traitement des requêtes en environnement volatile. Dans ce cas, exécuter une requête sur un mapping correspond à :

1. former le produit cartésien des instances des classes de base. Cela résulte en un ensemble de n-uplets, chacun étant formé par les instances des classes de base.
2. restreindre cet ensemble à l'aide de la contrainte simple et de la contrainte complexe associées au mapping.
3. pour chaque n-uplet de l'ensemble restreint, construire l'objet virtuel correspondant s'il n'existe pas dejà à l'aide de la fonction $base_v$.
4. Construire l'ensemble e' des objets virtuels associés à chaque n-uplet de l'ensemble restreint.

Le résultat de la requête est alors l'ensemble e'_m instance de $o_{qSetOfObject}$ contenant les objets virtuels élément de e' vérifiant la restriction de cette requête.

De manière plus formelle, cet algorithme correspond à l'exécution de la méthode induite par l'envoi d'un message $\lambda'(m, select, (r))$ avec m le mapping sur lequel est émise la requête r. $\lambda'(m, select, (r)) = e'_m$ tel que :

1. Soit $p = \bigotimes_{c \in base_m(m)} \widetilde{\pi}(c)$,
2. soit p' le résultat de l'application de $cont_s(m)$ et de $cont_c(m)$ sur p,
3. $\forall i \in p'$, s'il n'existe pas $o \in \widetilde{\pi}(m)$ tel que $base_v(o) = i$, il faut construire un tel objet.
4. Soit $e' = \{o \in \widetilde{\pi}(m) \mid \exists i \in p', base_v(o) = i\}$,

5. $query(e'_m) = r, elements(e'_m) = \{o \in e' \mid \lambda'(r, applyWhere, (o)) = o_{true}\}$.

Notons que si l'on suppose que les règles d'héritage définies dans la sous-partie précédente sont vérifiées, alors le résultat de la requête prend en compte l'ensemble des objets virtuels éléments de l'extension virtuelle de la classe c. En effet, la définition de l'ensemble e' permet de considérer l'ensemble des instances directes ou indirectes de chaque mapping actif et donc l'ensemble des instances virtuelles directes ou indirectes de la classe c.

Mappings en environnement volatile et peristant Lorsque l'on se place dans un cadre plus général, c'est-à-dire lorsque l'on permet la définition de mappings au dessus de classes de relais, il est alors nécessaire de modifier cet algorithme pour prendre en compte les données persistantes. En effet, même si l'algorithme fonctionne dans le cas général, il impose de charger toutes les données persistantes en mémoire pour effectuer la requête, ce qui n'est certainement pas optimal. La modification de l'algorithme précédent va donc consister à limiter la taille de l'ensemble construit à l'étape 1 en ne considérant que les objets de base vérifiant une certaine restriction déduite de la restriction de la requête et de la contrainte simple associée au mapping.

Pour ce faire, nous suivrons l'algorithme suivant, en considérant que la restriction d'une requête peut-être représentée sous la forme d'un arbre binaire dont les feuilles sont des expressions booléennes simples et les neuds les opérateurs booléens ET et OU[7]. Nous utilisons les mêmes notations que précédemment. En particulier, nous considérons un mapping noté m.

1. Soit f cet arbre. Pour chaque feuille de f, transformer les messages envoyés à des instance de m en messages envoyés sur les instances des classes de base à l'aide des correspondances.
2. Construire un arbre g à partir de $cont_s(m)$.
3. Construire un arbre binaire f', dont la racine est ET, la branche de gauche pointe sur la racine de f et la branche de droite porte sur la racine de g.
4. Pour chaque classe $c \in base_m(m)$, former la sous-restriction, notée r_c de la requête correspondant à cette classe à l'aide des règles suivantes :
   - Remplacer chaque feuille ne contenant pas d'envois de messages sur c par o_{true}.
   - Remplacer chaque feuille contenant des envois de messages sur plusieurs classes par o_{true}.
   - Si un nœud OU point sur o_{true}, le remplacer par o_{true}.
   - Si un nœud ET pointe sur o_{true} :
     - Si le deuxième élément du nœud est différent de o_{true}, remplacer le nœud par cet élément
     - Sinon le remplacer par o_{true}.
5. Pour chaque classe de base $c \in base_m(m)$, emettre sur c la requête basée sur la sous-restriction lui correspondant en envoyant le message $\lambda'(c, select, (r'))$ tel que $where(r') = r_c$ et $proj(r') = o_{Void}$. Le résultat de cette requête est un ensemble noté e_c
6. Former le produit cartésien des résultats de chaque requête émise sur les classes de base. En d'autres termes, $p = \bigotimes_{c \in base_m(m)} e_c$.

[7] Toute expression booléenne peut effectivement être mise sous cette forme.

Chapitre 4

Implantation en Java et Exemples

Pour l'implantation de nos modèles, nous souhaitons étendre un langage à objets largement reconnu et utilisé. Les langages à objets les plus répandus et utilisés actuellement sont sans aucun doute C++[Stroustrup93] et Java[Gosling et al.96]. Ce dernier permet de plus un prototypage rapide et possède une bibliothèque de réflexion minimale. C'est pourquoi nous l'avons choisi.

Ce choix n'est cependant pas sans conséquence sur l'architecture de notre système :

- D'une part, comme nous l'avons vu dans les chapitres précédents, nos modélisations sont basées sur des propriétés de réflexion et de spécialisation du langage, permettant de définir des objets transactionnels, des vues ou des mappings ou encore d'introduire dans le langage le concept d'envoi de message transactionnel. Or, ces propriétés sont en grande partie absentes de la bibliothèque de réflexion de Java. Celle-ci est plutôt conçue pour l'introspection des objets et des classes et au chargement dynamique de celles-ci.
- D'autre part, certains concepts de notre système tels que les comportements transactionnels doivent pouvoir être associés entre eux pour garantir des propriétés transactionnelles complexes. Les différentes associations possibles peuvent être obtenues aisément en implantant ces comportements sous la forme de classes et en les combinant par héritage multiple. Or Java est dépourvu d'héritage multiple.

Dans le chapitre 2, nous avons présenté le concept de réflexivité ainsi que certains des principaux langages le mettant en œuvre sous la forme de protocoles à métaobjets. Les langages décrits à cette occasion ont un point commun, ce sont tous des langages interprétés, c'est-à-dire dont les instructions peuvent être interprétées à la volée. La mise en œuvre d'une couche réflexive au sein de tels systèmes est, en général, assez naturelle car l'interprétation permet dans le même temps, de créer la structure de l'application, de définir son comportement et de l'exécuter. Dans les langages compilés, tels que Java, la phase de description de l'application est découplée de la phase d'exécution de celle-ci. En conséquence, les compilateurs qui leur sont associés sont très rarement définis de manière réflexive ce qui rend très difficile leur spécialisation. Il est néanmoins possible de définir des sytèmes basés sur des langages compilés, dont les propriétés sont proches de celles des systèmes réflexifs. Pour ce faire, trois approches sont possibles [Wu et al.97] :

- La première correspond à réaliser un précompilateur. Celui-ci se charge, à partir d'un code exprimé suivant une syntaxe étendue basée sur celle du langage considéré, de générer un code

standard pouvant être compilé par le langage, cette génération étant guidée par certaines expressions décrites par l'utilisateur suivant la même syntaxe que celle utilisée pour décrire l'application. En d'autres termes, en même temps que le développement de son application, le programmeur peut aussi décrire la manière dont elle doit être compilée.
- La deuxième est l'intégration d'un précompilateur, tel que défini ci-dessus, au compilateur originel. On obtient ainsi un compilateur réflexif. Cette approche est peu utilisée car elle nécessite la connaissance du compilateur, de ses sources, et du droit de le modifier.
- Enfin, la dernière correspond à réaliser un post-compilateur. Cette approche est l'approche symétrique de la première. Il s'agit non plus de transformer un code étendu en code standard, mais de transformer le résultat de la compilation d'un code standard en un résultat prenant en compte les spécifications de l'utilisateur. Bien que cette approche simule comme la première un compilateur réflexif, elle ne permet pas d'étendre la syntaxe de base du langage et donc sa sémantique selon les besoins de l'utilisateur.

En ce qui concerne le langage Java, citons trois systèmes mettant en œuvre respectivement la première, la deuxième et la troisième approche. Il s'agit de OpenJava [Tatsubori et al.98], MetaJava [Golm97] et Javassist [Chiba98]. Ces systèmes ont un point commun, ils permettent de spécialiser virtuellement ou non le compilateur suivant une sémantique Java, c'est-à-dire à l'aide de définitions de classes et de méthodes Java. Ils sont de plus tous trois définis de manière réflexive et mettent en œuvre un pseudo protocole à métaobjets comme nous le verrons plus loin pour le premier d'entre eux. Ils peuvent donc être utilisés pour faciliter l'implantation de notre système et résoudre une partie des problèmes soulevés précédemment.

Nous avons choisi d'utiliser OpenJava, c'est-à-dire la première approche, pour les raisons suivantes :

- Tout d'abord, il nous semble préférable de pouvoir étendre la syntaxe de Java pour permettre par exemple à l'utilisateur de notre système de spécifier le type d'envoi de message qu'il souhaite réaliser en même temps que la description de son application.
- D'autre part, nous souhaitons utiliser des systèmes de stockage de données externes. Or ceux-ci peuvent aussi utiliser des précompilateurs pour faciliter la gestion de la persistance des objets stockés en leur sein. Ces systèmes doivent être utilisés par le nôtre, ce qui implique que, lorsque l'on considère la chaîne d'exécution des différents outils nécessaires à la compilation, ces précompilateurs soient situés en aval de cette châine par rapport à notre système. Ceci implique que celui-ci soit aussi un précompilateur.

Dans la partie suivante, avant de présenter l'implantation de notre système en OpenJava, nous décrivons plus en détails celui-ci, à l'aide d'exemples extraits de [Tatsubori99].

Notons enfin que la majorité du travail réalisé sur OpenJava est basée sur des travaux similaires réalisés pour C++ (voir [Chiba95]).

4.1 OpenJava

OpenJava peut être vu comme un langage réflexif et spécialisable basé sur Java. La spécialisation du langage est effectuée à l'aide d'extensions à Java décrites à l'aide d'un programme dit de niveau méta, ou méta-programme, fourni lors de la compilation. Si aucun méta-programme n'est fourni, la syntaxe d'OpenJava et le comportement de son compilateur sont alors identiques à ceux de Java.

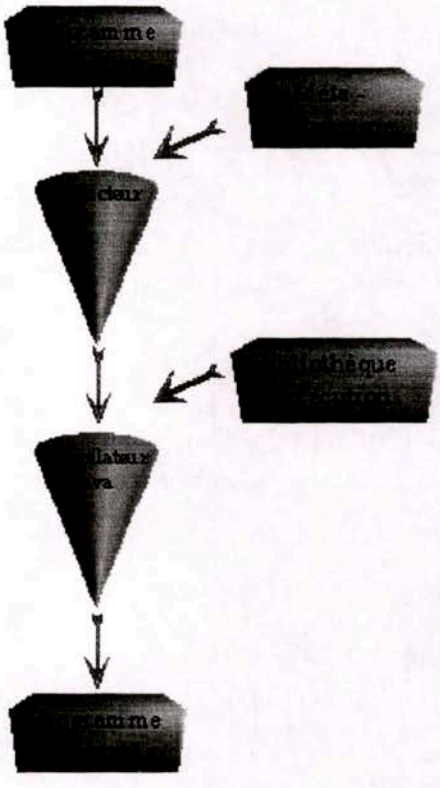


FIG. 4.1 – OpenJava

Un méta-programme permet d'étendre OpenJava à l'aide d'une interface appelée le protocole à métaobjets d'OpenJava. L'exécution du compilateur d'OpenJava se déroule en trois étapes : une étape de prétraitement, une étape de traduction de source OpenJava en source Java et une étape de compilation du source résultant. Le protocole à métaobjets d'OpenJava contrôle la deuxième étape. Il est défini en OpenJava, il peut être spécialisé et il permet de décrire comment un code OpenJava doit être traduit en code Java. Les classes définies dans un méta-programme guident la compilation des classes de l'application développée en OpenJava étendu par ce programme. Plus précisément, chaque classe d'une application écrite en OpenJava est compilée à l'aide d'une classe particulière du méta-programme. On peut donc, en ce sens, considérer que ces classes sont des métaclasses et que les classes de l'application sont leurs instances . Cependant, les classes d'une application développée en OpenJava sont in fine des classes Java standards et donc ne sont pas réellement instances de ces métaclasses. On dira plutôt que se sont des pseudo-instances de ces métaclasses.

En résumé, pour ajouter une extension à OpenJava, il faut décrire un programme de niveau méta permettant de guider la traduction. D'autre part, il peut souvent être nécessaire de fournir aussi une bibliothèque liée aux applications développées à l'aide de cette extension et permettant de l'utiliser. La figure 4.1 représente le processus de compilation d'un programme écrit en OpenJava. Notons que comme la dernière étape de ce processus est exécutée par un compilateur Java standard, le programme résultant peut être exécuté dans n'importe quelle machine virtuelle Java standard.

Dans la suite de cette partie, nous présentons un exemple simple d'extension d'OpenJava,

puis nous décrivons les différentes possibilités de ce système en termes de spécialisation de la compilation.

4.1.1 Exemple

Nous présentons ici une extension d'OpenJava permettant, lors de chaque appel d'une méthode dont la classe est une pseudo-instance d'une certaine métaclasse que nous appellerons `VerboseClass`, d'afficher un message.

Dans OpenJava, toute classe est par défaut pseudo-instance de la métaclasse de base d'OpenJava, nommée `OJClass` et toute métaclasse hérite d'`OJClass`. Cette métaclasse représente le métamodèle standard de Java. Toute classe est donc, comme nous l'avons déjà précisé, compilée par défaut comme une classe Java standard.

La programmation du protocole à métaobjets d'OpenJava se fait en trois étapes :

1. spécifier ce à quoi doit ressembler un programme OpenJava ;
2. spécifier ce en quoi celui-ci doit-être traduit ;
3. spécifier le méta-programme et la bibliothèque d'exécution associée.

Prenons l'exemple de la figure 4.2.

```
public class Hello instantiates VerboseClass {
   public static void main( String[] args ) {
      hello() ;
   }
   static void hello() {
      System.out.println( "Hello, world." ) ;
   }
}
```

FIG. 4.2 – Exemple de programme OpenJava

Le code présenté dans cette figure suit la syntaxe Java standard excepté pour la première ligne qui précise que la classe `Hello` est une pseudo-instance de `VerboseClass`, ce qui signifie que la compilation de la classe `Hello` est guidée par `VerboseClass`.

Définissons maintenant ce à quoi devrait ressembler le code résultant de la traduction de la classe `Hello` par OpenJava. Nous souhaitons faire afficher un message lors de chaque appel d'une méthode de cette classe. Un moyen de réaliser cela est de rajouter une instruction affichant ce message au début de chaque méthode. Le programme résultant, écrit cette fois en Java standard est présenté dans la figure 4.3.

Enfin, nous présentons dans la figure 4.4 le méta-programme, étendant OpenJava, permettant de réaliser la traduction ad-hoc.

Notons que comme nous l'avons précisé précédemment, `VerboseClass` hérite de `OJClass`.

```
public class Hello {
   public static void main( String[] args ) {
      System.out.println( "main is called." );
      hello();
   }
   static void hello() {
      System.out.println( "hello is called." );
      System.out.println( "Hello, world." );
   }
}
```

FIG. 4.3 – Exemple de programme généré par OpenJava

```
import openjava.mop.*;
import openjava.ptree.*;
public class VerboseClass instantiates Metaclass extends OJClass
{
   public void translateDefinition() throws MOPException {
      OJMethod[] methods = getDeclaredMethods();
      for (int i = 0; i < methods.length; ++i) {
         Statement printer = makeStatement(
           "System.out.println( \"" + methods[i] +
           " is called.\" );"
         );
         methods[i].getBody().insertElementAt( printer, 0 );
      }
   }
}
```

FIG. 4.4 – Exemple de programme de niveau méta

Pour modifier la traduction du code source, il suffit de surcharger la méthode `translateDefinition()`. Par défaut, cette méthode ne fait rien, ce qui correspond au cas où le code source est du code Java standard. Dans notre cas, nous souhaitons ajouter une instruction au début de chaque méthode des classes traduites à l'aide de `VerboseClass`.

L'algorithme correspondant est implanté dans le corps de la méthode `translateDefinition()`.

4.1.2 Fonctionnalités

Lors de la définition d'une métaclasse, c'est-à-dire, lors de la définition d'une sous-classe d'`OJClass`, il est possible d'effectuer quatre tâches distinctes :

- Définir des méthodes qui spécialisent la traduction des déclaration du programme utilisateur. Nous avons vu précédemment une méthode permettant de spécialiser la traduction de la définition d'une classe. OJClass contient aussi les méthodes permettant de spécialiser :

  - l'instanciation ;
  - l'accès aux champs en lecture ;
  - l'accès aux champs en écriture ;
  - l'appel d'une méthode.
- Utiliser des méthodes qui retournent des informations sur la classe en cours de traduction.
- Rajouter ou supprimer des déclarations de la classe en cours de traduction. En particulier, il est possible de rajouter ou de supprimer des champs, des méthodes, des constructeurs et de modifier la superclasse ou la liste des interfaces.
- Rajouter des mots-clés à la syntaxe du langage. Comme ces mots-clés sont définis au sein d'une métaclasse, ils ne peuvent être utilisés que lors de la description des classes pseudo-instances de cette métaclasse.

4.2 Transactions

Nous considérons ici uniquement les transactions en environnement volatile. Notons que nous ne décrirons pas dans ce chapitre l'implantation des métaclasses et des bibliothèques d'exécution de notre système c'est-à-dire l'étape 3 de la programmation en OpenJava, qui correspond à l'implantation des algorithmes présentés dans le chapitre précédent, mais uniquement les étapes 1 et 2 à l'aide d'exemples, ce qui permettra au lecteur de se familiariser avec la syntaxe de notre système.

4.2.1 Classes de gestion transactionnelle et Envoi de message transactionnel

```
class nom instanciates TransactionAbleClass
[extends superclasse]
[implements interface1, ...] {
   //définition Java standard des champs
   ...
   //définition des méthodes pouvant faire l'objet d'envois de message
transactionnel
   [modifier1 ...] type nom(arg1, ..., argn) [throws exception1, ...] {}
   //définition des méthodes ne pouvant faire l'objet d'envois de message
transactionnelle
   untransactional [modifier1 ...] type nom(arg1, ..., argn) [throws
exception1, ...] {}
}
```

FIG. 4.5 – Canevas de définition des classes de gestion transactionnelle

Pour mémoire, seules les méthodes, désignées par l'utilisateur et définies dans une classe de gestion transactionnelle, instance de `TransactionAbleClass`, peuvent faire l'objet d'envois de message transactionnel. Lorsque c'est le cas, un envoi de message transactionnel correspondant à une méthode m se réduit à :

1. démarrer une transaction T ;
2. démarrer un thread chargé d'exécuter m ;
3. attendre la fin de la transaction T ;
4. renvoyer le résultat de la méthode transactionnelle si T a été validée.

Notons que le point 1 peut consister à démarrer une transaction, une sous-transaction imbriquée ou une sous-transaction de haut-niveau. Plus précisément, l'utilisateur choisit, lorsqu'il envoie le message transactionnel, le type de transaction qu'il souhaite associer à ce message. S'il choisit un envoi de message transactionnel, que nous nommerons standard, alors s'il n'existe pas de transaction courante, une nouvelle transaction est démarrée, sinon, une sous-transaction imbriquée de la transaction courante est démarrée. S'il choisit un envoi de message transactionnel de haut niveau, alors s'il n'existe pas de transaction courante, une nouvelle transaction est démarrée, sinon, une sous-transaction de haut-niveau est démarrée.

Rappelons que notre système peut mettre en œuvre un autre type de sous-transaction, il s'agit des sous-transactions de thread. Une telle sous-transaction est automatiquement démarrée lorsque l'utilisateur ou le système, dans le cadre d'un envoi de message transactionnel, démarrent un thread dans l'environnement d'une transaction.

Enfin, la notion de transaction courante est fortement liée, dans notre système, à la notion de thread, chaque thread étant associé à un moment donné à au plus une transaction, ou sous-transaction.

```
class TATest instanciates TransactionAbleClass {
   void method1 (int a, int b) {
      ...
   }
   untransactional double method2 () {
      ...
   }
}

TATest t = new TATest() ;
t.transactional_method1(2,3) ;
t.hltransactional_method1(4,5) ;
t.transactional_method2() ; // non authorisé, erreur de compilation
t.method2() ;
```

FIG. 4.6 – Exemple de définition d'une classe de gestion transactionnelle

En résumé, nous souhaitons :

1. permettre à l'utilisateur de notre système de définir des classes dont les méthodes peuvent faire l'objet d'envois de message transactionnels ;
2. permettre à l'utilisateur de spécifier quel type d'envoi de message il souhaite envoyer pour une méthode donnée ;

3. permettre à l'utilisateur de spécifier quelles méthodes d'une classe de gestion transactionnelle peuvent faire l'objet d'un tel message.

En ce qui concerne le point 1, nous avons défini une métaclasse OpenJava nommée `TransactionAbleClass`, chargée de traduire la description des classes transactionnelles. En d'autres termes, chaque classe transactionnelle doit être une pseudo-instance de `TransactionAbleClass`.

Pour implanter la partie de notre système liée au point 2, nous devons faire face à une limitation d'OpenJava qui ne permet pas de redéfinir la syntaxe de l'envoi de message. En conséquence, il est impossible de définir un nouveau mot-clé permettant à l'utilisateur de spécifier le type d'envoi de message qu'il souhaite employer. Pour résoudre ce problème, lors de la traduction par `TransactionAbleClass` d'une méthode m pouvant faire l'envoi d'un message transactionnel, sont créées deux nouvelles méthodes m_1 et m_2 dont le type de retour et les arguments sont identiques à ceux de m et dont le nom est préfixé soit de `transactional_`, soit de `hltransactional_`. Dans le premier cas, la méthode générée correspond à un envoi de message transactionnel standard sur m. Dans le second cas, elle correspond à un envoi de message transactionnel de haut-niveau sur m. Dans les deux cas, le corps de ces méthodes contient les instructions nécessaires pour d'une part, créer et gérer la transaction ad-hoc, et d'autre part, démarrer un thread chargé d'exécuter m.

```
class TATest implements TransactionAbleObject {
   void method1 (int a, int b) {
      ...
   }
   double method2 () {
      ...
   }
   void transactional_method1 (int a, int b) throws TransactionalAbortException
{
      ... // gérer l'envoi de message transactionnel standard
   }
   void hltransactional_method1 (int a, int b) throws TransactionalAbortException
{
      ... // gérer l'envoi de message transactionnel de haut niveau
   }
}
```

FIG. 4.7 – Exemple de traduction d'une classe de gestion transactionnelle

Enfin, pour prendre en compte le point 3, nous avons défini un nouveau mot-clé nommé `untransactional`. Ce mot-clé peut être utilisé par l'utilisateur pour modifier la description d'une méthode définie dans une classe pseudo-instance de `TransactionAbleClass`. Un envoi de message transactionnel sur une méthode d'une telle classe n'est alors possible que si sa définition n'est pas préfixée par ce mot-clé.

Les figure 4.5 et 4.6 présentent respectivement le canevas de définition d'une classe de contrôle transactionnel et un exemple d'une telle définition.

Dans notre exemple, la méthode `method1` peut ainsi faire l'objet d'un envoi de message transactionnel, ce qui n'est pas le cas de la méthode `method2`.

Pour finir, la figure 4.7 présente le résultat de la compilation de la classe `TATest`. Cette classe implémente l'interface `TransactionAbleObject` pour respecter les règles de compatibilité de métaclasses que nous nous sommes fixés dans le chapitre 2. Notons qu'il suffit que `TransactionAbleObject` soit une interface car les règles de compatibilité définies précédemment ne portent que sur les méthodes. Enfin, comme cela est représenté sur la figure 4.7, les méthodes exécutées lors d'un envoi de message transactionnel lancent une exception lorsque la transaction qui leur est associée est annulée. Cette exception permet à l'utilisateur de connaître, s'il le souhaite, l'état final de la transaction.

4.2.2 Classes transactionnelles

```
// Note : c doit être une classe de comportement transactionnel associé à nom
class nom instanciates TransactionalClass
[extends superclasse]
[implements interface1, ...]
defaultBehavior c {
   //définition des méthodes identique à une classe de contrôle transactionnel
   ...
   //définition des champs transactionnels
   [modifier1, ...] type nom [=valeur] ;
   //définition des champs non transactionnels
   untransactional [modifier1, ...] type nom [=valeur] ;
}
```

FIG. 4.8 – Canevas de définition des classes transactionnelles

Notre système transactionnel permet aux objets transactionnels, instances des classes transactionnelles de participer activement à la gestion des propriétés transactionnelles telles que le contrôle de concurrence ou l'atomicité. Pour mémoire, une classe transactionnelle est une instance de la métaclasse `TransactionalClass`. Les objets transactionnels ne contiennent néanmoins pas de fonctionnalités transactionnelles propres si ce n'est la capacité de répondre aux messages transactionnels standard `abort`, `commit`, `begin` et `prepare`. Par le biais d'un mécanisme de transmission de messages (nous prenons ici ces termes au sens large, à savoir qu'un envoi de message peut correspondre aussi bien à l'accès en lecture ou écriture à un champs ou à l'exécution d'une méthode), les objets transactionnels délèguent la vérification et la gestion de leurs propriétés transactionnelles à d'autres objets dits comportements transactionnels que nous présentons dans la sous-partie suivante.

Lorsqu'un message est envoyé à un objet transactionnel lors d'une transaction, celui-ci est transmis au comportement transactionnel courant qui lui est associé et qui se charge du traitement transactionnel. Pour laisser la plus grande liberté d'optimisation, nous souhaitons permettre à l'utilisateur de spécifier pour chaque classe transactionnelle d'une part les messages pouvant être transmis au comportement transactionnel et, d'autre part, un type de comportement transactionnel par défaut.

```
class Compte instanciates TransactionalClass
defaultBehavior CompteAtomicLocking {
   double solde ;
   untransactional Statistique stat ;
   untransactional boolean isOk() {return solde>0 ;}
}
class Statistique instanciates TransactionalClass
defaultBehavior StatistiqueLocking {
   int op ;
   int opOK ;
}
...
cpt1.solde+=100 ;
cpt1.stat.op+=1 ;
if (cpt1.isOk()) cpt1.stat.opOK+=1 ;
else throw RuntimeAbortException ;
```

FIG. 4.9 – Exemple de définition et d'utilisation de classes transactionnelles

En résumé, nous souhaitons :

1. permettre à l'utilisateur de notre système de définir des classes transactionnelles ;
2. permettre à l'utilisateur de spécifier quels envois de message doivent prendre part à la gestion transactionnelle des instances de telles classes ;
3. permettre à l'utilisateur de spécifier un comportement transactionnel par défaut ;
4. permettre à l'utilisateur de modifier à la volée le comportement transactionnel d'un objet ;
5. permettre aux objets transactionnels de recevoir les messages `abort`, `commit`, `begin` et `prepare`.

Pour traiter le point 1, nous avons défini une nouvelle métaclasse OpenJava nommée `TransactionalClass` dont les pseudo-instances sont les classes transactionnelles.

Pour prendre en compte le point 2, le mot-clé `untransactional` peut être utilisé par l'utilisateur pour modifier la description d'une méthode ou d'un champ défini dans une classe transactionnelle. L'envoi d'un message à un objet transactionnel n'est alors transmis au comportement transactionnel qui lui est associé que si la définition de la méthode ou du champ correspondant n'est pas préfixée du mot-clé `untransactional`.

En ce qui concerne le point 3 nous avons introduit un autre mot-clé, nommé `defaultBehavior`. Ce mot-clé peut être utilisé par l'utilisateur lors de la définition d'une classe transactionnelle à la manière du mot clé Java `extends`. Dans ce cas, le nom suivant ce mot-clé correspond au comportement transactionnel par défaut des instances de cette classe.

Le point 4 induit l'ajout à toute classe transactionnelle d'une méthode permettant d'effectuer le changement. Notons que, comme nous l'avons déjà précisé, un tel changement n'est possible qu'en dehors de toute transaction.

```
class Compte implements TransactionalObject
{
   double solde ;
   Statistique stat ;
   boolean isOk() {return solde>0 ;}
   double jv_getSolde() {
      return jv_bo.getSolde(solde) ;
   }
   void jv_setSolde(double solde) {
      this.solde=jv_bo.setSolde(solde) ;
   }
   void abort() { //idem commit, begin, prepare
      jv_bo.abort() ;
   }
   BehaviorObject jv_bo = new CompteAtomicLocking(this) ;
}
```

FIG. 4.10 – Exemple de traduction d'une classe transactionnelle

Enfin, le point 5 nécessite l'ajout à toute classe transactionnelle des méthodes correspondant à l'interface transactionnelle standard. Notons que, comme seul le comportement transactionnel associé à un objet transactionnel connaît l'algorithme à suivre pour valider, annuler, ..., le corps de ces méthodes ne fait qu'exécuter la méthode correspondante sur le comportement transactionnel courant.

Les figures 4.8 et 4.9 présentent respectivement le canevas de définition d'une classe transactionnelle et un exemple d'une telle définition.

```
cpt1.jv_setSolde(cpt1.jv_getSolde()+100) ;
cpt1.stat.jv_setOp(cpt1.stat.jv_getOp()+1) ;
if (cpt1.isOk()) cpt1.stat.jv_setOpOK(cpt1.stat.jv_getOpOK()+1) ;
else throw RuntimeAbortException ;
```

FIG. 4.11 – Exemple de traduction d'expression d'utilisation d'objets transactionnels

Dans notre exemple, nous avons défini deux classes. La première, nommée `Compte`, représente un compte banquaire contenant deux champs, `solde` et `stat` et une méthode `isOK`. Un comportement transactionnel par défaut est spécifié à l'aide du mot-clé `defaultBehavior`. Cette expression précise que tout objet compte est géré par défaut de manière atomique et à l'aide d'un contrôle de concurrence par verrouillage. Notons que le champs `stat` et la méthode `isOk` ne sont pas pris en compte dans la gestion transactionnelle d'un objet compte comme cela est spécifié à l'aide du mot-clé `untransactionnal`.

La seconde classe, nommée `Statistique`, représente les statistiques de modification d'un compte. En particulier chaque instance de `Statistique` comptabilise le nombre d'opérations et le nombre d'opérations réussies effectuées sur le compte auquel elle est associée. Notons qu'il n'y a pas de raison de gérer les instances de `Statistique` de manière atomique, c'est

```
class Statistique implements TransactionalObject
{
   int op;
   int opOK;
   double jv_getOp() {
      return jv_bo.getOp(op);
   }
   void jv_setOp(double op) {
      this.op=jv_bo.setSolde(op);
   }
   double jv_getOpOK() {
      return jv_bo.getOpOK(opOK);
   }
   void jv_setOpOK(double opOK) {
      this.opOK=jv_bo.setSolde(opOK);
   }
   void abort() { //idem commit, begin, prepare
      jv_bo.abort();
   }
   BehaviorObject jv_bo = new StatistiqueLocking(this);
}
```

FIG. 4.12 – Classes transactionnelles - Exemple - Traduction (2)

pourquoi le comportement par défaut de ces objets est `StatistiqueLocking`.

Enfin, intéressons nous au code inscrit en dessous de ces deux classes. Nous supposerons qu'il s'exécute dans le cadre d'une transaction, par exemple, qu'il fait partie d'une méthode pouvant faire l'objet d'un envoi de message transactionnel. La dernière ligne de ce code indique que si le solde du compte est inférieur à 0, alors la transaction courante doit être annulée, ce qui aura notamment pour effet de remettre automatiquement le solde du compte dans l'état précédent le début de la transaction, celui-ci étant géré de manière atomique.

Pour terminer cette sous-partie, nous présentons dans les figures 4.10, 4.12 et 4.11 le résultat de la précompilation du code présenté dans la figure 4.9.

4.2.3 Classes de Comportements Transactionnel

```
//Note : M est sous-classe de BehaviorClass
class nom instanciates M
forClass nom2 {
   // définitions supplémentaires éventuelles (dépendent de M)
}
```

FIG. 4.13 – Canevas de définition des classes de comportement transactionnel

Pour mémoire, une classe de comportement transactionnel est une classe instance de `BehaviorClass` telle que définie dans notre modèle formel. Un comportement transactionnel est un objet instance d'une classe transactionnelle. Chaque classe de comportement transactionnel c est associée à une classe transactionnelle. Un comportement transactionnel, instance de c, peut être associé à un objet instance de cette classe et définit alors les propriétés transactionnelles vérifiées par celui-ci.

Notons que la description d'une classe de comportement transactionnel dépend des propriétés transactionnelles que l'on souhaite lui attribuer. Par exemple, lors de la définition d'une classe de comportement transactionnel atomique, il doit être possible de définir des méthodes inverses de celles définies dans la classe transactionnelle associée, comme cela a été présenté dans le chapitre 3. En conséquence, chaque propriété transactionnelle doit être mise en œuvre à l'aide d'une métaclasse de comportement transactionnel particulière. Néanmoins, un certain nombre de fonctionnalités sont communes à toutes ces métaclasses. En particulier celles permettant à l'utilisateur :

1. de définir des classes dont les instances seront des comportements transactionnels ;
2. de spécifier la classe transactionnelle associée à chaque classe de comportement transactionnel.

```
class CompteAtomicLocking instanciates AtomicLockingBehavior
forClass Compte {
}
class StatistiqueLocking instanciates LockingBehavior
forClass Statistique {
}
```

FIG. 4.14 – Exemple de définition de classes de comportement transactionnel

Pour traiter le point 1 nous avons défini une nouvelle métaclasse OpenJava, nommée `BehaviorClass`, comme racine d'héritage des métaclasses de comportement transactionnel. Notre système dispose, lors de la rédaction de ce mémoire, des métaclasses de comportemement transactionnel nommées `AtomicBehaviorClass`, `LockingBehaviorClass` et `AtomicLockingBehaviorClass`, mettant en œuvre respectivement les propriétés transactionnelles d'atomicité, de contrôle de concurrence et une combinaison de ces deux propriétés.

Pour prendre en compte le point 2, nous avons ajouté le mot-clé `forClass` à la syntaxe d'OpenJava. Ce mot-clé doit être utilisé par l'utilisateur lors de la définition d'une classe de comportement transactionnel à la manière du mot clé Java `extends`. Le nom suivant `forClass` correspond alors à la classe transactionnelle qui lui est associée.

La figure 4.13 présente le canevas général de définition des classes de comportement transactionnel.

Pour finir, les figures 4.14, 4.15 et 4.16 présentent la description et la traduction des classes de comportement transactionnel utilisées dans les définitions des classes `Compte` et `Statistique` de l'exemple introduit dans la sous-partie précédente.

```
class CompteAtomicLocking implements BehaviorObject
{
   double getSolde(double solde) {
      ...
   }
   double setSolde(double solde) {
      ...
   }
   void abort() { //idem commit, begin, prepare
      ...
   }
   Compte myObject ;
}
```

FIG. 4.15 – Exemple de traduction d'une classe de comportement transactionnel

4.2.4 Travaux connexes

Pour terminer cette partie, nous présentons différents travaux portant sur l'intégration de systèmes transactionnels au sein de langages de programmation. Nous comparons ces travaux aux nôtres en portant une attention particulière à ceux appliqués au langage Java.

4.2.4.1 Argus.

Argus [Liskov88] est un langage de programmation et un système dédié au développement et à l'exécution de programmes distribués. Ce système permet, par couplage fort, la gestion de concurrence et la garantie d'exécution atomique à travers le concept, nommé par l'auteur, d'action, correspondant au concept de transaction imbriquée vérifiant les propriétés A et I. Argus fournit un ensemble de types de base atomiques et permet à l'utilisateur d'en définir d'autres. A l'instar de nos travaux, l'atomicité et le contrôle de concurrence ne portent que sur un sous-ensemble des objets manipulés par une action, objets dits atomiques. De plus, Argus propose une sémantique transactionnelle implicite. Contrairement à nos travaux, cependant, le comportement des objets est fixé dès leur création et ne peut être que de deux sortes : soit aucun comportement transactionnel spécifique, soit un comportement atomique par versionnement couplé à un contrôle de concurrence par verrouillage.

4.2.4.2 Arjuna et Java Arjuna.

De la même manière que précédemment, ces travaux se basent sur le concept d'action et permettent la gestion d'objets soit à comportements transactionnels (uniquement atomicité et verrouillage), soit sans (voir par exemple [Parrington92], [Little et al.98] et [Little et al.98b]). Ces systèmes ont pour but, à l'instar d'Argus, de faciliter la programmation d'applications distribuées en C++ (Arjuna) ou Java (Java Arjuna). Cependant, à l'inverse d'Argus et de nos travaux, le début (begin), la fin des actions (commit ou abort), ainsi que la pose de verrous doivent être explicitement inscrits dans le code de l'application par le programmeur. Ces

```
class StatistiqueLocking implements BehaviorObject
{
   double getOp(double op) {
      ...;
   }
   double setOp(double op) {
      ...;
   }
    double getOpOK(double opOK) {
      ...;
   }
   double jv_setOpOK(double opOK) {
      ...;
   }
   void abort() { //idem commit, begin, prepare
      ...;
   }
   Statistique myObject;
}
```

FIG. 4.16 – Exemple de traduction d'une classe de comportement transactionnel (2)

systèmes ne proposent donc pas de sémantique transactionnelle implicite. L'approche utilisée ici est, comme Argus, une approche par couplage fort.

4.2.4.3 PJava.

Pjava (voir [Daynes95], [Daynes et al.97] et [Atkinson et al.96]) est un langage de programmation persistant basé sur Java. Le système transactionnel de cet outil est donc, par définition, intégré par couplage fort et ne permet pas l'utilisation de systèmes transactionnels externes. PJava permet de spécifier le type de comportement associé à chaque objet comme dans les travaux décrits précédemments. Néanmoins, la technique de verrouillage peut être raffinée, ce qui permet une plus grande souplesse et une plus grande spécialisation du comportement transactionnel. L'utilisation du système transactionnel est similaire à l'utilisation d'un système transactionnel d'un SGBD classique à travers JDBC. Une transaction est un objet sur lequel on peut envoyer les messages begin, end et commit. Comme il n'y a pas de gestion explicite des verrous comme dans Arjuna, ce système est donc plus proche de nos travaux en ce qui concerne la propriété de sémantique transactionnelle implicite sans pour autant avoir complètement intégré cette sémantique au monde objet (il n'y a pas d'envoi de messages transactionnels). Enfin, ce système repose sur une modification de la machine virtuelle Java ce qui supprime l'un des aspects fondamentaux de Java : l'indépendance de l'architecture d'exécution, la portabilité.

4.2.4.4 Extension persistante de SML.

Le but de ces travaux (voir par exemple [Haines et al.93], [Haines et al.94] et [Wing et al.92]) est de fournir au langage Standard ML un système transactionnel par couplage fort pour faciliter la persistance des entités du langage. L'originalité de ce système est la factorisation des propriétés transactionnelles. L'exécution d'une transaction est l'exécution d'une suite de fonctions imbriquées, chaque fonction garantissant une certaine propriété transactionnelle pour l'ensemble des entités qu'elle manipule. Les auteurs définissent quatre types de fonctions : la persistance (D), la possibilité annulation(A), la gestion des threads(I) et le verrouillage(C). La combinaison de ces quatre fonctions revient à exécuter une transaction avec les propriétés ACID, bien que chaque fonction puisse être exécutée indépendamment. Dans ce système, la sémantique transactionnelle est implicite puisqu'il s'agit juste d'exécuter une fonction. Cependant, il n'est pas possible d'appliquer des comportements transactionnels différents de ceux prédéfinis. De plus, toute entité manipulée au sein d'une transaction a le même comportement : celui défini par l'imbrication des fonctions.

4.3 Requêtes

Comme précédemment, nous ne considérons ici que les requêtes en environnement volatile. Pour mémoire, une requête est représentée au sein de notre système par deux chaînes de caractères, exprimées dans le langage hôte, qui spécifient respectivement :

- une expression de restriction basée sur une expression conditionnelle ;
- une expression de projection.

```
class nom instanciates QueryAbleClass
[extends superclasse]
[implements interface1, ...]{
   //définition Java standard
}
```

FIG. 4.17 – Requêtes : canevas de définition

Nous considérons de plus que l'exécution d'une requête correspond à l'envoi d'un message *select* à une classe instance de `QueryAbleClass`, telle que définie dans notre modèle formel, et que le résultat de la requête contient la projection, calculée en fonction de la clause de projection associée à la requête, de l'ensemble des instances directes ou indirectes de cette classe vérifiant la restriction associée à la requête.

En conséquence, nous souhaitons :

1. permettre à l'utilisateur de notre système de préciser les classes pouvant faire l'objet de requêtes ;
2. permettre à l'utilisateur d'envoyer un message *select* à ces classes ;
3. permettre à l'utilisateur de préciser les clauses de projection et de restriction sous forme d'expressions java ;

4. faire en sorte que le gestionnaire de requêtes traite, lors d'une requête, l'ensemble des instances directes ou indirectes de la classe concernée par la requête.

```
class Employe instanciates QueryAbleClass {
   String nom ;
   String prenom ;
   long numss ;
   Projet proj ;
}
class Projet {
   String nom ;
   double budget() {...}
}
...
JVList Employe.select("nom+prenom","proj.budget()>100000") ;
...
```

FIG. 4.18 – Requêtes : exemple

Pour traiter le point 1, nous avons défini une nouvelle métaclasse OpenJava nommée `QueryAbleClass`, dont les pseudo-instances peuvent faire l'objet de requêtes.

Le point 2 nécessite, lors de la traduction d'une classe pseudo-instance de `QueryAbleClass`, d'ajouter à cette classe une méthode statique correspondant au message *select*.

Le point 3 nécessite la mise en œuvre d'un interprète d'expressions Java. Cet interprète à été réalisé à l'aide d'une part du générateur de parseur SableCC [Gagnon99] pour la construction des arbres d'évaluation associés à chaque clause de requête, et d'autre part, de la bibliothèque de réflexion Java pour l'évaluation de ces arbres. Notons qu'une conséquence de cette aproche est que le corps de la méthode `select` correspond uniquement à invoquer cet interprète.

```
class Employe implements QueryAbleObject {
   String nom ;
   String prenom ;
   long numss ;
   Projet proj ;
   static JVList select (String proj, String restr) {...}
   JVSet jv_extension ;
}
```

FIG. 4.19 – Requêtes : exemple de traduction

Le point 4 nécessite, pour chaque classe pseudo-instance de `QueryAbleClass`, la connaissance de son extension. Ceci est implanté à l'aide d'un champ de type ensemble représentant cette extension. Celui-ci est automatiquement ajouté à chaque classe pouvant faire l'objet d'une requête sous la forme d'une variable statique. Lors de l'instanciation d'une telle classe, l'objet créé est automatiquement ajouté à l'extension de sa classe. Notons que, du fait d'une gestion

automatique de la mémoire en Java, cet objet ne peut être détruit que si l'utilisateur décide explicitement de le retirer de l'ensemble représentant l'extension de sa classe.

Les figures 4.17 et 4.18 présentent respectivement le canevas de définition d'une classe pouvant faire l'objet de requêtes et un exemple d'utilisation de notre gestionnaire de requêtes.

Pour terminer cette partie, nous présentons, dans la figure 4.19, le résultat de la traduction de la classe `Employe`.

4.4 Classes de relais

```
//Note : M doit être sous-classe de ProxyClass
// Note 2 : c doit être une classe de comportement transactionnel associé
à nom
class nom instanciates M
baseName nB
entityName nE
user nU
password p
idExpression {exp}
[extends superclass]
[implements interface1, ...]
[defaultBehavior c]
// définitions supplémentaires, dépendent de M {
   //définition standard Java
}
```

FIG. 4.20 – Canevas de définition des classes de relais

Pour mémoire, les classes de relais sont les instances de la métaclasse `ProxyClass` telle que définie au sein de notre modèle formel. Les relais sont les instances des classes de relais. Notons que `ProxyClass` sous-classe `TransactionalClass`, les relais pouvant être associée de même que les objets transactionnels aux comportements transactionnels.

Dans notre modèle formel, `ProxyClass` sous-classe aussi `QueryAbleClass`, les relais devant pouvoir faire l'objet de requêtes. En Java, l'héritage est simple, ce deuxième sous-classement n'est donc pas possible. Les comportements propres à la gestion des requêtes doit donc être réimplanté dans `ProxyClass`. Notons que cette gestion est néanmoins très différente de celle définie dans `QueryAbleClass` puisqu'il est nécessaire de traduire les requêtes émises sur les classes de proxies en requêtes compréhensibles par les sources de données externes. L'impact de l'héritage simple de Java sur l'implantation de notre système est donc, dans une certaine mesure, négligeable en ce qui concerne la gestion des relais.

Le concept de relais est associé au concept de persistance et permet d'accéder aux sources de données externes au langage. Les bibliothèques d'accès à ces sources ont été formalisées précédemment. Nous en avons développé trois qui correspondent respectivement aux SGBD Poet ([Poet]), Versant ([Versant]) et ObjectDriver, ce dernier étant un SGBD objet virtuel au dessus de bases de données relationnelles.

```
class Compte instanciates ObjectDriverPersistantClass
defaultBehavior CompteAtomicLocking
baseName demo
minCache 10
maxCache 20
idExpression{compteID}
entityName PCompte {
   int minCache=10 ;
   int maxCache=20 ;
   double solde ;
   Statistique stat ;
   untransactional boolean isOk() {return solde>0 ;}
}
class Statistique instanciates ObjectDriverPersistantClass
defaultBehavior StatistiqueLocking
baseName demo
entityName PStat
idExpression {statId}
{
   int op ;
   int opOK ;
}
...
JVList result=Compte.select(null,"solde>50") ;
Compte cpt1=result.first() ;
cpt1.solde+=100 ;
cpt1.stat.op+=1 ;
if (cpt1.isOk()) cpt1.stat.opOK+=1 ;
else throw RuntimeAbortException ;
```

FIG. 4.21 – Exemple de définition d'une classe de relais

Dans notre modèle tel que définit dans le chapitre précédent, les relais sont gérés à l'aide d'une politique de cache. En d'autres termes, un relais ne contient pas de données mais uniquement une interface d'accès aux données. En conséquence, la définition d'un relais implique en Java la génération par notre système de deux classes :

- une classe, dont les instances sont des relais, décrivant l'interface d'accès, une référence sur les données et un identificateur unique de ces données ;
- une classe pour les données, décrivant la structure de ces données. A l'exécution, chaque relai est associé, lorsque ses données sont chargées, à une instance de la classe de données qui lui correspond. L'ensemble des instances de cette classe forment le cache associé au relais.

De plus, pour mettre en œuvre notre politique de cache en Java, il est nécessaire de prendre en compte le type de gestion de la mémoire de ce langage. La mémoire étant gérée automatiquement, il est en particulier impossible de détruire explicitement un objet[1]. En conséquence, le

[1]Notons qu'il est possible de demander au système de libérer les objets non référencés en invoquant expli-

déchargement d'un objet du cache consiste uniquement, au sein de notre système, à supprimer la référence à cet objet du relai auquel il est associé. Comme l'utilisateur n'a pas connaissance de cet objet, il ne devrait y avoir aucune autre référence sur cet objet, et il devrait donc être supprimé de la mémoire automatiquement. Néanmoins, d'une part, il est impossible d'être sur de ce fait et d'autre part, la suppression n'est dans tous les cas pas immédiate. Il faut donc s'attendre à ce que notre politique de gestion de cache ne soit pas très précise lorsque implantée en Java.

En résumé, l'implantation du concept de relai en Java doit :

1. permettre à l'utilisateur de spécifier précisément l'entité représentée par la classe de relais qu'il souhaite décrire ;
2. permettre à l'utilisateur de spécifier le comportement transactionnel par défaut des relais ;
3. permettre à l'utilisateur de préciser la politique de cache telle que définie dans le chapitre précédent.

```
class Compte implements ObjectDriverPersistentObject
{
   boolean isOk() {return solde>0 ;}
   double jv_getSolde() { //idem pour stat
      ...
   }
   void jv_setSolde(double solde) { //idem pour stat
      ...
   }
   void abort() { //idem commit, begin, prepare
      ...
   }
   static JVList select(string proj, string restr) {
      return JVCompteData.select(proj,restr) ;
   }
   BehaviorObject jv_bo = new CompteAtomicLocking(this) ;
   JVCompteData jv_data = null ;
   Object id ;
}
```

FIG. 4.22 – Exemple de traduction d'une classe de relais

Le point 1 concerne la description du SGBD ou de la source de données utilisée et dépend de celui-ci. Néanmoins, il est possible que pour accéder aux données stockées, il soit nécessaire de connaître le nom de l'entité représentée par la classe de relais, la manière d'identifier les données, le nom de la base dans laquelle celles-ci sont stockées ainsi que les informations de sécurité telle qu'un nom d'utilisateur et un mot de passe. En conséquence, nous avons défini

citement le « ramasse-miette ». Néanmoins, il n'est pas certain que le résultat de cette exécution consiste en la suppression de l'objet que l'on souhaite détruire.

une métaclasse OpenJava, nommée `ProxyClass` dont les pseudo-instances sont des classes de relais. Nous avons aussi défini les métaclasses, sous-classes de `ProxyClass` permettant l'accès aux SGBD Poet, Versant et ObjectDriver et nommées respectivement `PoetPersistantClass`, `VersantPersistantClass` et `ObjectDriverPersistantClass`. Enfin, nous avons introduit dans la syntaxe OpenJava, les mots-clés `baseName`, `entityName`, `idExpression`, `user` et `password` qui permettent de spécifier respectivement :

- le nom de la base ;
- le nom de l'entité représentée ;
- une expression permettant d'identifier les données ;
- le nom de l'utilisateur et son mot de passe.

De plus, pour éviter d'avoir à définir un nom d'utilisateur et un mot de passe pour chaque classe de relais, l'utilisateur de notre système peut aussi définir ces informations de manière globale à son application, ou pour chaque base utilisée.

L'implantation de la partie de notre système correspondant au point 2 est identique à celle décrite pour les classes de contrôle transactionnel.

Enfin, le point 3 nécessite, lors de la traduction de chaque classe de relai, d'une part de créer une classe de données telle que décrite précédemment, et d'autre part, d'introduire dans la classe de relais les références nécessaires au bon fonctionnement de ceux-ci. Par ailleurs, la mise en œuvre du cache nécessite la connaissance du nombre maximum d'objets pouvant être contenus dans le cache, du nombre minimum d'objets du cache à décharger et de la méthode donnant une indication sur les possibilités de déchargement des objets du cache. Nous imposons que ces trois informations puissent être renseignées par l'utilisateur lors de sa description des relais par le biais des mots-clés `minCache` et `maxCache`, utilisables de la même façon que le mot-clé `extends`, et par la méthode booléenne nommée `unloadable`.

```
class JVCompteData {
   double solde ;
   Statistique stat ;
   double jv_getSolde() { //idem pour stat
      return jv_proxy.jv_bo.getSolde(solde) ;
   }
   void jv_setSolde(double solde) { //idem pour stat
      solde=jv_proxy.jv_bo.setSolde(solde) ;
   }
   Compte jv_proxy ;
   static JVList select(string proj, string restr) {
      ...
   }
}
```

FIG. 4.23 – Exemple de traduction d'une classe de relais (2)

La figure 4.20 présente le canevas de définition des classes de relais.

Pour finir, reprenons l'exemple des classes `Compte` et `Statistique` telles que définies dans

la partie précédente, en supposant cette fois que ces deux classes représentent des classes de relais dont les données sont stockées dans le SGBD ObjectDriver. (voir la figure 4.21)

Notons tout d'abord qu'une fois les classes de relais définies, celles-ci s'utilisent alors comme n'importe quelle classe transactionnelle, en particulier en ce qui concerne la définition et l'utilisation de classes de comportement transactionnel. Dans cet exemple, si l'on a utilisé les mots-clés `baseName` et `entityName` pour préciser l'origine des données, nous n'avons indiqué aucune information de sécurité. Dans ce cas, le système utilise le nom et le mot de passe de l'utilisateur défini pour la base de données correspondante et s'ils n'existent pas, ceux définis globalement. Enfin, notons qu'il n'est défini aucune valeur de cache pour la classe `Statistique`, l'utilisateur laissant au système la possibilité d'utiliser des valeurs par défaut.

Les figures 4.22 et 4.23 présentent le résultat de la traduction de la classe `Compte`.

Pour terminer cette partie, notons que l'ajout de la persistance au sein d'un langage de programmation tel que Java a fait l'objet de nombreux travaux (voir par exemple les travaux de [Morrison et al.89], [Morrison et al. 96] et [Srinivasan97]). A la différence de notre approche, ces travaux se concentrent sur les techniques de stockage des objets d'une application et non sur l'exploitation de sources de données indépendantes et/ou hétérogènes.

4.5 Vues

```
class nom implements VirtualClass
[extends superclass]
[implements interface1, ...]
{
   //définition Java standard sans constructeurs
   ...
}
```

FIG. 4.24 – Canevas de définition des classes virtuelles

Pour terminer ce chapitre, intéressons nous à l'implantation des vues. Comme nous l'avons introduit dans le chapitre 2, le concept de vue est représenté au sein de notre système à la fois par les classes virtuelles et par les mappings. Nous avons aussi choisi de permettre uniquement la définition de vues non matérialisées c'est-à-dire dont les données sont calculées lors de chaque accès.

De manière plus précise, lorsque l'utilisateur de notre système souhaite définir une vue, il doit :

1. définir la structure et le comportement des classes virtuelles formant la vue ;
2. déterminer les entités, formant la base de la vue (le terme d'entité regroupe ici, tous les types de classes pouvant faire l'objet de requêtes, ce qui inclue les classes virtuelles) ;
3. définir les correspondances entre les classes virtuelles et les entités de base de la vue par le biais des mappings.

Pour le point 1, la définition d'une classe virtuelle implique de définir ses champs et ses méthodes. Notons que comme nous ne souhaitons pas matérialiser les vues, la structure finale

```
class nom implements MappingClass
structBy c
baseName c1, ..., cn
baseVar v1, ..., vn
[extends superclass]
[implements interface1, ...]
[baseDelete vk, ..., vm]
[constraintBy {expr}] {
   // définition de méthodes Java standards
   ...
   //définition des algorithmes de calcul
   get nom_c() {...}
   //définition des algorithmes de mise à jour
   set nom_c() {...}
   //définition de la contrainte complexe
   nom() {...}
}
```

FIG. 4.25 – Canevas de définition des mappings

d'une classe virtuelle en tant qu'entité Java ne doit contenir aucun des champs Java correspondant à la définition, mais seulement des accesseurs particuliers référençant les algorithmes de calcul et/ou de mise à jour des champs tels que définis dans le mapping associé à chaque objet virtuel. Pour ce faire, nous avons défini la métaclasse `VirtualClass` dont les pseudo-instances sont des classes virtuelles, telles que, la traduction d'une classe virtuelle résulte en une classe telle que définie ci-dessus.

Le point 3 correspond à donner la possibilité à l'utilisateur de notre système de définir des correspondances entre une classe virtuelle et des entités de base à l'aide d'un mapping. Celui-ci contient à la fois les algorithmes de calcul et de mise à jour des instances de la classe virtuelle qui lui est associée et la définition des contraintes simples et complexes telles que présentées dans le chapitre précédent. Au sein de notre système, un mapping doit être défini comme pseudo-instance de `MappingClass`. Notons qu'un mapping doit être associé à la fois à la classe virtuelle qu'il met en correspondance et aux entités de base qu'il utilise. Pour ce faire, nous introduisons trois nouveaux mots clés associés à `MappingClass` : `structBy`, `baseName` et `baseVar`. Ces trois mots-clés permettent de définir respectivement la classe virtuelle que l'on souhaite mettre en correspondance, les noms des entités de base utilisées, et les variables utilisées dans la description du mapping pour référencer ces entités. De plus, nous imposons que les algorithmes de calcul et de mise à jour des instances de la classe virtuelle soient définis dans des méthodes sans argument. Chacune de ces méthodes doit avoir comme nom celui du champ qu'elle représente, et son type de retour doit être `get` si elle définit un algorithme de calcul et `set` dans le cas contraire. Nous imposons aussi que les contraintes simple et complexe soient définies respectivement à l'aide du mot-clé `constraintBy` et sous la forme d'un constructeur par défaut Java. Enfin, nous permettons à l'utilisateur d'utiliser le mot-clé `deleteBase` pour préciser les variables correspondant aux objets de base à détruire lorsqu'un objet virtuel est supprimé. Notons que si ce mot-clé n'est pas utilisé, le système impose que

tous les objets de base soient détruits, lorsqu'un objet virtuel est supprimé.

```
class Emp instanciates ObjectDriverPersistantClass
baseName demo
entityName EMP
idExpression {ss} {
   long ss;
   double sal;
}
```

FIG. 4.26 – Vues - classe de base

```
class EmployeView instanciates ViewClass {
   String nom;
   double salaire;
}
...
EmployeView.addMapping(new MappingView());
JVList result=EmployeView.select("nom","salaire>1000");
EmployeView emp = result.first();
System.out.println(emp.nom);
...
```

FIG. 4.27 – Exemple de définition et d'utilisation d'une classe virtuelle

Les figures 4.24 et 4.25 présentent respectivement le canevas de définition des classes virtuelles et le canevas de définition des mappings.

Pour terminer, prenons un exemple. Soient les entités de base suivantes :

- `Emp`, une classe de relais (voir la figure 4.26) ;
- `Employe`, telle que présentée dans la figure 4.18.

Nous souhaitons définir une vue de ces deux entités permettant de connaître le nom et le salaire de chaque employé. Pour ce faire, il est possible d'utiliser la classe virtuelle présentée dans la figure 4.27 associée au mapping présenté dans la figure 4.28. Notons qu'il est ensuite possible d'effectuer des requêtes sur la classe `EmployeView` de la même manière que sur n'importe quelle classe pouvant faire l'objet de requêtes.

Enfin, nous présentons dans les figures 4.29 et 4.30 le résultat de la traduction des classes `EmployeView` et `EmployeMapping`. Notons que la traduction d'EmployeView génère une classe correspondant à la classe virtuelle et une interface permettant de référencer les mappings correspondant à cette classe et de leur envoyer des messages.

```
class EmployeMapping instanciates MappingClass
structBy EmployeView
baseName Employe, Emp
baseVar e1, e2
constraintBy {e1.ssnum==e2.ss} {
   get nom() {return e1.nom;}
   get salaire() {return e2.salaire;}
}
```

FIG. 4.28 – Exemple de définition d'un mapping

```
class EmployeView implements ViewObject {
   String getNom() { // idem salaire
      return mapping.getNom();
   }
   void setSS(String nom) { // idem salaire
      mapping.setNom(nom);
   }
   JV_EmployeMapping mapping;
   static JV_List select(String proj, String restr) {...}
   static void addMapping(JV_EmployeMapping map) {...}
   static void removeMapping(JV_EmployeMapping map) {...}
...
}
interface JV_EmployeMapping {
   String getNom(); //idem salaire
   vois setNom(String nom); // idem salaire
}
...
System.out.println(emp.getNom());
...
```

FIG. 4.29 – Exemple de traduction d'une classe virtuelle

```
class EmployeMapping implements MappingObject, JV_EmployeMapping {
   String getNom() { ..corps défini dans get nom().. } //idem salaire
   void setNom() {..renvoie une erreur (set nom() n'est pas définie)..}
//idem salaire
   static JV_List select(String proj, String restr) {...}
   Employe e1;
   Employe e2;
...
}
```

FIG. 4.30 – Exemple de traduction d'un mapping

Conclusion

Dans ce mémoire de thèse, nous avons proposé des solutions aux problèmes liés à l'intégration – structurelle et partielle – de sources de données hétérogènes.

A la différence de travaux antérieurs, nous avons choisi non seulement d'uniformiser l'accès à de multiples sources de données, persistantes ou non, mais aussi de rendre cet accès complètement paramétrable. En d'autres termes, nous avons chercher à résoudre conjointement les problèmes liés à l'hétérogénéité d'accès aux données et ceux liés à l'hétérogénéité de stockage des données.

Après une étude des différents modèles et systèmes proposés dans la littérature, et après une analyse des besoins en termes d'accès homogènes et paramétrables aux données, nous avons conclu qu'un système couplant de manière mixte un langage à objets et des outils de stockage de données résoud les problèmes cités précédemment.

Dans le but de modéliser et de mettre en œuvre un tel système nous avons étudié les principes d'intégration des concepts de transaction, de relai, de requête et de vue au sein d'un langage à objets.

Pour réaliser cette étude, nous nous sommes intéressés plus particulièrement aux langages réflexifs à objets, langages ouverts et paramétrables, en présentant un modèle formel d'un tel langage. Au cours de la description de ce modèle, nous avons défini formellement les concepts d'objet, de classe, de métaclasse, d'héritage, de méthode, de typage et d'envoi de message ainsi que des propriétés de compatibilité de classes et de métaclasses.

Nous avons par la suite étendu ce formalisme pour y inclure les concepts de transaction, de relai, de requête et de vue.

Durant cette étude, nous avons présenté un nouveau modèle transactionnel ouvert et paramétrable adapté à la programmation par objets. Ce modèle repose d'une part sur la séparation des différentes composantes transactionnelles, chaque objet pouvant posséder son propre comportement transactionnel, et d'autre part, sur le concept d'envoi de message transactionnel. Le système mettant en œuvre, en particulier, ce modèle est présenté dans le dernier chapitre de ce mémoire. Il permet à la fois d'accéder selon les besoins de ses utilisateurs à de multiples systèmes transactionnels hétérogènes, et de définir aisément de nouveaux modèles transactionnels.

Nous avons aussi décrit un système de gestion de requêtes paramétrable et uniforme adapté non seulement à l'exploitation des données persistantes par le biais de relais, mais aussi à la gestion des données volatiles. Nous avons de plus présenté un nouveau système paramétrable de vues basé sur les concepts indépendants de classe virtuelle et de mapping. Ce système permet de représenter de manière uniforme et selon les structures définies par l'utilisateur des

données volatiles et/ou des données persistantes éventuellement stockées dans des sources de données hétérogènes.

Pour finir, nous avons souhaité que les différentes modélisations présentées dans ce mémoire soient aussi générales que possible, afin de permettre leur implantation au sein de nombreux langages. A titre d'exemple, nous avons présenté une implantation de nos modèles en Java. Pour le développement de ce système, nous avons utilisé OpenJava, ce qui nous a permis de conserver une approche réflexive et de proposer un outil ouvert et paramétrable.

Perspectives. Au cours de ce mémoire, nous avons pu mettre à jour plusieurs thèmes d'étude pouvant être développés à la suite de nos travaux. En particulier, il nous semble intéressant d'étudier des extensions à nos modèles pour prendre en compte les données distribuées. En effet, comme nous avons pu le souligner au cours de nos discussions, les concepts de transaction ou de vue peuvent rendre la gestion des données distribuées plus aisée et donc faciliter la tâche du programmeur devant développer des applications distribuées.

Par ailleurs, notre modèle transactionnel nous paraît adapté à l'étude de nouveaux systèmes transactionnels dédiés à la gestion des données semi-structurées et en particulier à la gestion des données disponibles sur Internet. En effet, comme nous avons souhaité nos modélisations ouvertes et complètement paramétrables, il est assez facile de prendre en compte d'une part de nouveaux métamodèles et, d'autre part, de nouveaux modèles transactionnels aussi complexes soient-ils, sans pour autant devoir redéfinir toute une architecture transactionnelle.

Enfin, il nous parait prometteur d'essayer d'étendre nos modèles et nos algorithmes pour permettre d'une part d'effectuer des jointures entre plusieurs classes, et d'autre part, d'optimiser la gestion des requêtes, notamment en ce qui concerne la décomposition des requêtes globales et inversement l'assemblage des résultats des sous-requêtes.

Table des figures

Bibliographie

[Abiteboul et al.97] Abiteboul S., Cluet S. et Milo T. - *Correspondance and Translation for Heterogeneous Database*, dans les actes de ICDT'97, ed. par Afrati (F.N.) et Kolaitis (P.), Lecture Notes in Computer Science, vol. 1186, pp. 351-363. - Delphes, Grèce, janvier 1997.

[Agarval et al.92] Agarwal S., Keller A.M., Wiederhold G. et Saraswat K. - *Flexible Relation : An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases*, dans les actes du 7th International Conference on Data Engineering (ICDE'95), ed. par Yu (P.S.) et Chen (L.P.), pp. 459-504. - Taipei, Taiwan, mars 1995

[Ahmed et al.91] Ahmed R., De Smedt P, Du W., Kent W., Ketabchi M.A., Litwin W.A., Rafii A. et Shan M.C. - *The Pegasus Heterogeneous Multidatabase System*, IEEE Computer, vol. 24(12), pp. 19-27. - décembre 1991

[Amer-Yahia et al.97] Amer-Yahia S., Brèche P. et Souza dos Santos C. - *Object Views and Updates*, dans Ingénierie des Systèmes d'Information, vol. 5(1) - avril 1997.

[ANSI75] ANSI/SPARC/X3 Study Group on Database Management Systems. - *Interim Report on Database Management system*, CAEMA, ACM SIGMOD Bulletin, Washington D.C. - 1975

[ANSI78] ANSI/SPARC/X3 Study Group on Database Management Systems. - *Framework Report on Database Management system*, Information systems, vol. 3(3) - 1978

[APM91] APM - *ANSA Reference Manual*, APM Ltd. Poseidon House, Cambridge, UK. - 1991.

[Arbouy et al.94] Arbouy S., Bezos A, Cutting-Decelle A.F. Diakonoff P., Germain-Lacour P., Letouzey J.P. et Viel C. - *STEP : Concepts fondamentaux*, AFNOR. - 1994.

[Atkinson et al.96] Atkinson M.P., Jordan M.J., Daynès L. et Spence S. - *Design Issues for Persistent Java : a type-safe, object-oriented, orthogonally persistent system*, dans les actes du 7th International Workshop on Persistent Object Systems (POS-7), ed. par Connor R.C.H et Nettles S., pp. 33-47 - Cape May, New Jersey, U.S.A., mai 1996.

[Atkinson et al.96b] Atkinson M.P., Daynès L., Jordan M.J., Printezis T. et Spence S. - A*n Orthogonally Persistent Java*, ACM Sigmod Record, vol. 25(4), pp. 68-75. - décembre 1996.

[Attardi et al.89] Attardi G., Bonini C., Boscotrecase M.R., Flagella T. et Gaspari M. - *Metalevel Programming in CLOS*, dans les actes de 3rd European Conference on Object-Oriented Programming (ECOOP'89), ed. par Cook S., pp. 243-256. - Nottingham, UK, juillet 1989.

[Baldoni et al.95] Baldoni R. et Salza S. - *Deadlock Detection in Multidatabase Systems : a Performance Analysis*, Rapport de Recherche INRIA N°2668, 21 pages. - septembre 1995.

[Barghouti et al.91] Barghouti N.S. et Kaiser G.E. - *Concurrency Control in Advanced Database Applications*, ACM Computing Surveys, vol. 23(3), pp. 269-317. - septembre 1991.

[Bellahsene97] Bellahsene Z. - *Identifying Virtual Composite Objects : A Step Forward to Update Object Views*, dans les actes du 7th International Workshop and Expert system Applications (DEXA'97), ed. par Wagner R., pp. 523-528. - Toulouse, France, septembre 1997.

[Bernstein et al.87] Bernstein P.A., Hadzilacos V. et Goodman N. - *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, 352 pages - 1987.

[Bobrow et al.88] Bobrow D.G., DeMichiel L.G., Gabriel R.P., Keene S., Kiczales G. et Moon D.A. - Common Lisp Object System Specification, rapport technique ISO X3J13 Document 88-002R, SIGPLAN Notices, vol. 23, special issue. - septembre 1988.

[Boulanger et al.98] Boulanger D. et Dubois G. - *An object Approach for Information Systems Cooperation*, dans Information Systems, vol. 23 (6), pp. 383-399. - 1998.

[Bouraquadi et al.96] Bouraqadi-Saâdani N.M.N., Leroux T. et Rivard F. - *Safe Metaclass Programming*, dans les actes de ACM SIGPLAN Conference on Object-Oriented Programing Systems, Language and Applications (OOPSLA'98), SIGPLAN Notices, vol. 33(10), pp. 84-96. - Vancouver, Colombie Brittanique, Canada, octobre 1998.

[Breitbart et al.91] Breitbart Y., Litwin W. et Silbershatz A. - *Deadlock problems in a multidatabase transaction manager*, in Proceedings of IEEE COMPCON'91, pp. 145-151. - San Francisco, U.S.A., juillet 1991.

[Breitbart et al.95] Breitbart Y., Garcia-Molina H. et Silberschatz A., *Transaction Management in Multidatabase Systems*, dans Modern Database Systems, Wom Kim ed., Addison-Wesley Publishing Compagny (New-York), pp. 573-591. - New-York, U.S.A., 1995.

[Bright et al.94] Bright M.W., Hurson A.R. et Pakzad S. - *Automated resolution of Semantic Heterogeneity in Multidatabases*, dans ACM Transactions on Database Systems, vol. 19(2), pp. 212-253. - juin 1994.

[Bukhres et al.95] Bukhres O.A., Elmagarmid A.K. et Pitoura E. - *Object Orientation in Multidatabase Systems*, ACM Computing Surveys, vol. 27(2), pp. 141-195. - juin 1995.

[Busse et al.94] Busse R., Fankhauser P., Hulk G., Klas W. - *IRO-DB, An object-Oriented approach towards federated and interoperable DBMS*, dans les actes du 1st International Workshop on Advances in Databases and information Systems (ADBIS'94), pp. 178-186. - Moscou, Russie, mai 1994.

[Cattel et al.97] Cattell R.G.G., Barry D., Bartels D., Berler M., Eastman J., Gamerman S., Jordan D., Pringer A., Strickland H. et Wade D. (édité par). - *The Object Database Standard : ODMG 2.0*, Morgan Kaufmann Publishing, 288 pages. - 1997.

[Chen et al.96] Chen I.A., Kosky A., Markovitz V.M. et Szeto E. - *OPM*QS : The Object Protocol Model Multidatabase Query System*, Rapport de Recherche n° LBL-38181 Laurence Berkeley National Lab, 29 pages - 1996.

[Cheung et al.98] Cheung S. et Matena V. - Java Transaction API, Sun Microsystem Incorporated, V. 031, 42 pages. - juin 1998.

[Chiba95] Chiba S., *A Metaobject Protocol for C++*, dans les actes de 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), SIGPLAN Notices, vol. 30(10), pp. 285-299 - Austin, Texas, USA, octobre 1995.

[Chiba98] Chiba S. - *Javassit - A Reflection-based Programming Wizard for Java*, dans les actes du ACM SIGPLAN OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 5 pages - Vancouver, Colombie Brittanique, octobre, 1998.

[Codd70] Codd E.F. - *A Relational Model for Large Shared Data Banks*, Communication of the ACM, vol. 13(6), pp. 377-387. - juin 1970.

[Cointe87] Cointe P. - *Metaclasses are First Class : the ObjVlisp Model*, dans les actes de la Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), ed. par Meyrowitz N.K., SIGPLAN Notices, vol. 22(12), pp. 156-167. - Orlando, Floride, USA, décembre 1987

[Cosmadadis et al.84] Cosmadadis S.S. et Papadimitriou C.H. - *Updates of relational views*, in journal of the ACM, vol. 31(4), pp. 742-760. - 1984.

[Damodoran-Kamal et al.94] Damodaran-Kamal S.K., Francioni J.M. et Pissinou N. - *A Modeling Paradigm for Multidatabases*, dans les actes de l'International Conference on Parallel Processing - Chicago, USA, août 1994. .

[Daynes et al.95] Daynès L., Gruber O. et Valduriez P. - *Locking in OODBMS Client Supporting Nested Transactions*, dans les actes de 7th International Conference on Data Engineering (ICDE'95) , ed. par Yu P.S. et Chen A.L.P., pp. 316-323. - Taipei, Taiwan, mars 1995.

[Daynes et al.97] Daynès L. Atkinson M.P. et Valduriez P. - *Customizable concurrency control for Persistent Java*, 7ème chapitre de Advanced Transaction Models and Architectures, 1997 Edition Kluwer, Editions Jajadia S. et Keuchberg L., 33 pages - août 1997.

[Daynes95] Daynès L. - *Extensible Transaction Management in Pjava*, dans le 1st International Workshop on Persistent Java, 16 pages - Drynan, Ecosse, septembre 1995.

[Delobel et al.95] Delobel C. et Souza dos Santos C. - *Object Views of Relations*, dans les actes du congrès INFORSID, 18 pages - Grenoble, France, mai 1995.

[Detlef et al.88] Detlef D., Herlihy M., Wing J.M. - *Inheritance of Synchronization and Recovery Properties in Avalon/C++*, IEEE Computer, vol. 21(12), pp. 57-69. - décembre 1988.

[Dixon et al.94] Dixon M. et Kalmus J. - *Semantic Heterogeneous in Interoperating Databases an Investigation Based Upon Engineering Maintenance*, dans les actes d'ERCIM'94, pp. 93-107. - Barcelone, Espagne, 1994.

[Dubois97] Dubois G. - *Apport de l'intelligence artificielle à la coopération de systèmes d'information automatisés*, thèse de doctorat de l'université Jean Moulin Lyon III, Institut d'Administration des Entreprises, 219 pages - janvier 1997.

[Duwairi et al.96] Duwairi R.M., Fiddian N.J. et Gray W.A. - *A Multiple View Definition System for Supporting Interoperability among Heterogeneous and Autonomous Databases*, dans les actes du 10th ERCIM Database Research Group Workshop on Heterogeneous Information Management, pp. 33-42 - Prague, Czech, 1996.

[Eppinger et al.86] Eppinger J.L., Spector A.Z., Bloch J.J., Daniels D.S., Draves R., Duchamp D., Menees S.G., et Thompson D.S. - *The Camelot Project*, Database Engineering Bulletin, vol. 9(3), pp.23-34. - novembre 1986.

[Fankhauser et al.96] Fankhauser P., Finance B. et Klas W. - *IRO-DB : Making Relational and Object-Oriented Database Systems Interoperable*, dans les actes de 5th International Conference on Extending Database Technology (EDBT'96), ed. par Apers P.M.G, Bouzeghoub M. et Gardarin G., Lecture Notes in Computer Science, vol. 1057, pp. 485-489 - Avignon, France, mars 1996.

[Ferber89] Ferber J. - *Computational Reflection in Class based Object Oriented Languages*, dans les actes de la Conference on Object-Oriented Programming systems, Languages and Applications

(OOPSLA'89), ed. par Meyrowitz N.K., SIGPLAN Notices, vol. 24(10), pp. 317-326. - Nouvelle Orléans, Louisiane, USA, octobre 1989.

[Fernandez et al.89] Fernandez M.F. et Zdonik S.B. - *Transaction Groups : A Model for Controlling Cooperative Transactions*, dans les actes du 3rd International Workshop on Persistent Object Systems (POS'89), ed. par Rosenberg J. et Koch D., pp. 341-350. - Newcastle, New South Wales, janvier 1989.

[Florescu et al.95] Florescu D., Raschid L. et Valduriez P. - *Query Reformulation in Multidatabase Systems using Semantic Knowledge*, Rapport de Recherche INRIA n° 2561, 29 pages. - mai 1995.

[Gagnon99] Gagnon E., SableCC, *An Object-Oriented Compiler Framework*, université McGill, Montréal, http ://www.sable.mcgill.ca/~gagnon/sablecc/thesis.html - mars 1999.

[Gardarin89] Gardarin G. - *SGBD avancés*, editions Eyrolles, Paris, France, 255 pages. - décembre 1989.

[Gardarin93] Gardarin G. - *Maîtriser les bases de données*, editions Eyrolles, Paris, France, 347 pages. - mars 1993

[Gardarin95] Gardarin G., Sha F., Tang Z.H. - *Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System*, dans les actes du 22th International Conference on Very Large Data Bases (VLDB'96), ed. par Vijayaraman T.M., Buchmann A.P., Mohan C et Sarda N.L., pp. 378-389. - Bombay, Indes, septembre 1996.

[Gardarin99] Gardarin G. - *Les bases de données - objet et relationnel*, editions Eyrolles, 788 pages. - mars 1999.

[Gentile94] Gentile M. et Zicari R. - *Updating Views in Object Oriented Database Systems*, dans les actes de l'International Symposium on Advanced Database Technologies and Their Integration (ADTI'94) 12 pages - Nara, Japon, octobre 1994.

[Georgakopoulos91] Georgakopoulos D., Rusinkiewicz M. et Sheth A.P. - *On serializability of Multidatabase Transactions Through Forced Local Conflicts*, dans les actes ,du 7th International Conference on Data engineering, pp. 314-323. - Kobe, Japon, avril 1991.

[Goldberg et al.83] Goldberg A. et Robson D. - *SMALLTALK-80 : the Language and its Implementation*, Addison-Wesley Publishing Company, 714 pages. - 1983.

[Golm97] Golm M. - *Design and Implementation of a Meta Architecture for Java*, mémoire de Diplomarbeit im Fach Informatik, Friedlich-Alexander Univeristy, DA-14-002-97 - janvier 1997.

[Gosling et al.96] Gosling J., joy W et Steele G. - *The Java Language Specifications*, edité par Addison-Wesley Publishing Company, New-York, USA - 1996.

[Graham99] Graham I. - Introduction To HTML and URLs, publication Internet, http ://www.utoronto.ca/webdocs/HTMLdocs/NewHTML/intro.html. - janvier 1999.

[Graube89] Graube N. - *Metaclass Compatibility*, dans les actes de Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89), ed. par Meyrowitz N.K., pp. 305-315. - Nouvelle Orléans, Louisianne, U.S.A., octobre 1989.

[Gray et al.93] Gray J. et Reuter A. - *Transaction Processing : Concepts and Techniques*, Morgan kaufmann, 1070 pages. - 1993.

[Gray81] Gray J. - *The Transaction Concept : Virtues and Limitations*, dans les actes de 7th International Conference on Very Large Database (VLDB'81), pp. 144-154. - Cannes, France, septembre 1981.

[Guerraoui96] Guerraoui R., *Strategic Directions in Object Oriented Programming*, dans ACM Computing Surveys, vol. 28(4), pp. 691-700. - juin 1996.

[Haines et al.93] Haines N., Kindred D., Morriset J.G., Nettles S.M. et Wing J.M., *Tinkertoy Transactions*, rapport de recherche CMU-CS-93-202, Carnegie Mellon University, Pittsburgh, 19 pages - décembre 1993.

[Haines et al.94] Haines N., Kindred D., Morriset J.G., Nettles S. et Wing J.M. - *Composing First-Class Transactions*, dans ACM Transactions on Programming Languages and Systems, Short Communication, vol. 16(6), pp. 1719-1736. - novembre 1994.

[Harder et al.93] Härder T. et Rothermel K. - *Concurrency Control Issues in Nested Transactions*, dans Very Large Data Base Journal, vol. 2(1), pp. 39-74. - janvier 1993.

[Ingalls78] Ingalls D.H.H. - *The SMALLTALK-76 Programming System Design and Implementation*, in Proceedings of the 5th Annual ACM Symposium on the Principles of Programming Languages (POPL'78), pp. 9-17. - Tucson, Arizona, 1978.

[ISO86] International Standard Organization - *Information processing - text and office systems -standard generalized markupl language (sgml)*, ISO 8879 :1986(E), ISO/IEC 10744. - 1986.

[ISO90] International Standard Organization - Information Technology - Open Systems Interconnection - Commitment, Concurency and Recovery, ISO/IEC 9804/9805. - 1990

[ISO95] International Standard Organization, ISO-ANSI Working Draft : *Database Language SQL (SQL3)*, ed. par Jim Melton, rapport technique ISO X3H2-95-083/DBL :YOW-004, mars 1995.

[Jautzy97] Jautzy O. - *Protocoles à métaobjets : Une Base Formelle pour leur introduction dans les systèmes multibases*, rapport de recherche 97-114, CERMICS, 21 pages - novembre 97.

[Jautzy99a] Jautzy O. - *Highly customizable transaction management in Java*, dans les actes du SDPS World Conference on Integrated Design Process and Technology (IDPT'99), Society for Design and Process Science, 9 pages - à paraître.

[Jautzy99b] Jautzy O. - *Gestion personnalisée de transactions en Java*, prix du jeune chercheur INFORSID 1999, dans les actes du XVIIème Congrès Informatique des Organisations et Systèmes d'Information et de Décision (INFORSID'99), pp. 331-350. - La Garde, France, juin 1999.

[Jautzy99c] Jautzy O. - *Interoperable databases : a Programming Language Approach*, dans les actes de L'International Database Engineering and Applications Symposium (IDEAS'99), ed. par Desai B.C. et Grahne G., pp. 63-71. - Montreal, Canada, août 1999.

[Jeffery et al.90] Jeffery K.G., Hutchinson L., Kalmus J., Wilson M., Berendt W. et Macnee C. - *A Model for Heterogeneous Distributed Database Systems*, dans les actes du 12th British National conference on Databases (BNCOD 12), ed. par Bowers D.S., pp. 221-234 - Guildford, United Kingdom, juillet 1990.

[Kaiser93] Kaiser G.E. et Pu C. - *Dynamic Restructuration of Transactions*, dans Transaction Models for Advanced Applications, Data Management systems, Ed. par Elmagarmid A., Morgan Kaufman (San Mateo). - 1993.

[Kapitsaia et al.97] Kapitsaia O., Tomasic A. et Valduriez P., *Dealing with Discrepancies in Wrapper Functionnality*. Rapport de Recherche INRIA n° 3138, 21 pages. - 1997.

[Kapsammer et al.97] Kapsammer E. et Wagner R. - *The IRO-DB Approach : Processing Queries in Federated Database Systems*, dans les actes du 8th International Workshop on Database and Expert Systems Applications (DEXA'97), pp. 713-718. - Toulouse, France, septembre 1997.

[Kelley et al.95] Kelley W., Gala S., Kim, W., Reyes T. et Graham B. - *Schema Architecture of the UNISQL/M Multidatabase Architecture*, dans Modern Database Systems, Addison Wesley publishing Company, ed. par Kim W., pp. 621-648. - 1995.

[Kickzales et al.91] Kickzalès G., des Rivières J. et Bobrow D.G. - *The Art of the Metaobject Protocol*, The MIT Press, 335 pages. - 1991.

[Kim95a] Kim W. - *Technologiy for Interoperating Legary Databases*, dans Modern Database Systems, Addison-Wesley Publishing Compagny (N.Y.), ed. par Kim W., pp. 515-519. - 1995.

[Kim95b] Kim W. - *Next-Generation Database Technology*, dans Modern Database Systems, Addison-Wesley Publishing Compagny (N.Y.), ed. par Kim W., pp. 5-13. - 1995.

[Kleewein96] Kleewein J. - *Practical Issues With Commercial Use of federated Databases*, dans les actes du 22th International Conference on

Very Large Data Base (VLDB'96), ed. par Vijayaraman T.M., Buchmann A.P., Mohan C. et Sarda N.L., page 580. - Bombay, Indes, septembre 1996.

[Klas et al.96a] Klas W., Fankhauser P. et Muth P., - *Database Integration using the Open Object oriented Database System VODAK*, dans Object Oriented Multidatabase Systems : A solution for Advanced Applications, ed. par Kim W., Prentice Hall, pp. 472-532. - 1996.

[Klas et al.96b] Klas W. et Schrefl M. - *Metaclasses and Their Application Data Model Tailoring and Database Integration*, Lecture Notes in Computer Science n°943. - 1996.

[Kung et al.81] Kung H.T. et Robinson J.T. - *On Optimistic Methods for Concurrency Control*, dans ACM Transactions on Database Systems, vol. 6(2), pp. 212-226. - juin 1981.

[Kuno et al.93] Kuno H.A. et Rundersteiner E.A. - *The MultiView OODB View System : Design and Implementation*, dans Theory and Practice of Object systems, vol. 2(3), pp. 202-225. - 1996.

[Lamport94] Lamport L. - *Latex : A Document Preparation System - 2nd Edition*, Addison-Wesley Publishing Company, 272 pages. - 1994.

[Lebastard93] Lebastard F. - *DRIVER : Une couche objet virtuelle peristante pour le raisonnement sur les bases de données relationnelles*, thèse de doctorat, Institut National des Sciences Appliquées de Lyon (INSA), 380 pages. - mars 1993.

[Lipton et al.90] Lipton R.J. et Naughton J.F. - *Query Size estimation by Adaptative Sampling*, dans les actes du 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90), pp. 40-46. - Nashville, Tenesse, avril 1990.

[Liskov88] Liskov B., *Distributed Programming in Argus*, dans Communications of the ACM, vol. 31(3) , pp. 300-312. - mars 1988.

[Little et al.98] Little M.C. et Shrivastava S.K. - *Understanding the Role of Atomic Transactions and Group Communications in Implementing Persistent Replicated Objects*, dans les actes du 8ème International Workshop on Persistent Object Systems : Design Implementation and use, 12 pages - Triburan, Californie, U.S.A., août 1998.

[Little et al.98b] Little M.C. et Shrivastava S.K. - *Java Transaction for the Internet*, dans la 4th Conference on object Oriented Technologies and systems (COOTS'98), 12 pages - avril 1998.

[Litwin et al.90] Litwin W., Mark L. et Roussopoulos N. - *Interoperability of Multiple Autonomous Databases*, dans ACM Computing Surveys, vol. 22(3), pp. 267-293 - septembre 1990

[Llirbat et al.96] Llirbat F., Simon E. et Tombroff D. - *Using Versions in Update Transactions*, rapport de recherche INRIA n°2940, 41 pages - juillet 1996.

[Lynch83] Lynch N.A. - *Multilevel Atomicity : A new correctness criterion for database concurrency control*, dans ACM Transactions on Database Systems, vol. 8(4), pp. 484-502 - décembre 1983.

[Machuca et al.94] Machuca F., Gannouni S. et Smahi V. - *Using an Intermediate Functional Language to Federate Heterogeneous Databases*, dans les actes d'ERCIM'94, pp. 58-74 - 1994.

[Madria97] Madria S.K. - *Timestamp-based Approach for the Detection and Resolution of Mutual Conflicts in Distributed Systems*, dans les actes du 8th International Workshop on Database and Expert Systems Applications (DEX'97), ed. par Wagner R., pp. 692-669. - Toulouse, France, septembre 1997.

[Mak96] Mak N.W. - *Interfacing Heterogenous Databases with a formal specification of the attribute mappings*, rapport de recherche de OTAN SHAPE Technical Centre - mars 1996.

[Mehrotra93] Mehrotra S., Rastogi R., Breitbart Y., Korth H.F. et Silbershatz A. - *Efficient Global Transaction Management in Multidatabase Systems*, dans les actes du 3rd International Conference on Database systems for Advanced Applications (DASFAA), ed. par Moon S.C. et Ikeda H., pp. 29-36. - Daejon, Korée, avril 1993.

[Mehrotra91] Mehrotra S., Rastogi R., Korth H.F., et Silbershatz A. - *Non-Serializable Executions in Heterogeneous Distributed Database Systems*, dans les actes du 1st International Conference on Parallel and Distributed Information Systems (PDIS 1991), pp. 245-252. - Miami Beach, Floride, USA, décembre 1991.

[Mendhekar et al.93] Mendhekar A. et Friedman D.P. - *Towards a Theory of Reflective programming language*, dans les actes d'OOPLA'93 Workshop on Reflection and Metalevel Architectures, 11 pages - 1993.

[Meng et al.95] Meng W. et Yu C., *Query Processing in Multidatabase Systems*, dans Modern Database Systems, ed. par Kim W., Addison Wesley Publishing Company (New-York), pp 551-572. - 1995.

[Morrison et al.89] Morrison R. et Atkinson M.P. - Persistent Systems Architectures, dans les actes du 3rd International Workshop on Peristent Object systems (POS'89), ed. par Rosenberg J. et Koch D., pp. 73-97. - Newcastle, New South Wales, janvier 1989.

[Morrison et al. 96] Morrison R., Connor R., Kirby G. et Munro D. - *Can Java Persist ?*, dans les actes du 1st international Workshop on persistence for Java, 15 pages - 1996

[Motro87] Motro A. - *Superviews : Virtual Integration of Multiple Databases*, IEEE Transactions on Software Engineering, vol. 13(7), pp. 785-798. - juillet 1987.

[Moss85] Moss J.E. - *Nested Transactions : An approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Massachusset. - 1985.

[Navathe96] Navathe S.B. - *A schema integration Facility Using Object Oriented Data Model*, dans Object Oriented Multidatabase Systems : A solution for advanced applications, ed. par Bukhres O.A. et Elmagarmid A.K., Prentice Hall, pp. 105-128. - 1996.

[OMG97] Object Management Group. - *Transaction service specification v. 1.1*, OMG Document, http ://www.omg.org, 90 pages. - novembre 1997.

[OSF91] Object Software Foundation. - *OSF DCE V1.X Requirements*, OSF DCE Reliable Group. - 1991.

[Oszu et al.99] Öszu M.T. et Valduriez P. - Principles of Distributed Database Systems, Second Edition, Prentice Hall - 1999.

[Paepcke90] Paepcke A. - *PCLOS : Stress Testing CLOS - Experiencing the Metaobject Protocol*, dans les actes de OOPSLA-ECOOP'90 Conference on Object-Oriented Programming systems, Languages and Applications - European Conference on Object-Oriented Programming, ed. par Meyrowitz N., SIGPLAN Notices, vol. 25(10), pp. 194-211. - Ottawa, Canada, octobre 1990.

[Parrington et al.88] Parrington G.D. et Shrivastava S.K, *Implementing Concurrency control in Reliable Distributed Object-Oriented Systems*, dans les actes de ECOOP'88 European Conference on Object-Oriented Programming, ed. par Gjessing S. et Nygaard K., Lecture Notes in Computer Science, vol. 322, pp. 233-249. - Oslo, Norvège, août 1988.

[Parrington92] Parrington G.D. - *Programming Distributed Applications Transparently in C++ : Myth or Reality ?*, dans les actes de OpenForum 92 Technical Conference, pp. 205-218. - novembre 1992.

[Pitoura et al.95] Pitoura E., Bukhres O. A. et Elmagarmid A. K. - *Object Orientation in Multidatabase Systems*, dans ACM Computing surveys, vol. 27(2), pp141-195, - juin 1995.

[Poet] Poet, Système de gestion de Base de Données, http ://www.poet.com.

[Radeke et al.95] Radeké E., Sholl M.H. - *Fonctionality for Object Migration Among Distributed, Heterogeneous, Autonomous DBs*, RIDEDOM'95, pp.58-66. - 1995.

[Roantree et al.99] Roantree M., Skehill A. - *Constructing Multidatabase Collections Using an Extended ODMG Model*, dans les actes de 2nd workshop on Engineering Federated Information Systems (EFIS'99), ed. par Conrad S., Hasselbring W. et Saake G., pp. 95-106. - Kühlungsborn, Allemagne, mai 1999.

[Royer92] Royer J.C. - *A propos des concepts de CLOS*, dans les actes de JFLA91 : Langages Applicatifs, pp. 150-158. - 1991.

[Rukoz91] Rukoz M. - *La détection d'inerblocage dans les transactions imbriquées réparties*, dans Technique et Science Informatiques, pp. 448-454 - 1991.

[Saltor et al.91] Saltor F., Castellanos M. et Garcia-Solano M. - *On Canonical Models for federated DBs*, ACM Sigmod Record, vol. 20(4), pp. 44-48. - 1991.

[Saltor et al.94] Saltor F., Compolerrich B. et garcia-Solaco M. - *On Architecture For Federated Databases Systems*, dans les actes d'ERCIM'94, pp. 8-26 - 1994.

[Silbershatz et al.95] Silershatz A., Stonebrake M. et Ullman J., *Database Research : Achievements and opportunities into the 21st Century*, Rapport du NSF Workshop on the Future of Database Systems Research, 10 pages. - Mai 1995.

[Simeon99] Siméon J. - *Intégration de sources de données hétérogènes - Ou comment marier simplicité et efficacité*, thèse de doctorat, Université de Parix XI, 156 pages. - janvier 1999.

[Sheth et al.90] Sheth A. et Larson J. - *Federated Database Systems for Managing Distributed Heterogeneous and Autonomous Databases*, dans ACM Computing Surveys, vol. 22(3), pp 183-236. - septembre 1990.

[Souza dos Santos94a] Souza dos Santos C. - *Design and Implementation of an object-Oriented View Mechanism*, dans les actes des 10èmes Journées Bases de Données Avancées (BDA'94), Clermont-Ferrand, France. - septembre 1994.

[Souza dos Santos94b] Souza dos Santos C., Delobel C. et Abiteboul S. - *Virtual Schemas and Bases*, dans les actes de l'International Conference on Extending Data Base Technology, Cambridge. - mars 1994.

[Souza dos Santos95] Souza dos Santos C. - *Un mécanisme de vues pour les systèmes de gestion de bases de données objet*, thèse de doctorat, université de Paris-Sud, 191 pages - 1995.

[Spector85] Spector A.Z. - *The TABS Project*, dans Database Engineering Bulletin, vol. 8(2), pp. 19-25. - juin 1985.

[Spence96] Spence S. - *Distribution Strategies for Persistent Java*, dans les actes du 1st Workshop on Persistence and Java, 16 pages. - septembre 1996.

[Springsteel93] Springsteel F.N. - *Object Based Schema integration For heterogeneous Databases : A Logical Approach*, dans les actes de 4th International Conference on Database and Expert Systems Applications (DEXA'93), ed. par Marik V., Lazansky J. et Wagner R., pp. 166-180. - Prague, République Chèque, septembre 1993.

[Skeen81] Skeen D. - *Nonblocking Commit Protocols*, dans les actes du 1981 ACM SIGMOD Conference on Management of Data, ed. par Lien Y.E., pp. 133-142. - Ann Arbor, Michigan, USA, avril 1981.

[Srinivasan97] Srinivasan V. et Chang D.T. - *Object Persistence in Object-Oriented Applications*, IBM Systems Journal, vol. 36(1), 23 pages. - 1997.

[Staudt et al.96] Staudt M. et Jarke M. - *Incremantal Maintenance of Externally materialized views*, dans les actes du 22th International Conference on Very Large Data Bases (VLDB'96), ed. par Vijayaraman T.M., Buchmann A.P., Mohan C. et Sarda N.L., pp. 75-86. - Bombay, Indes, septembre 1996.

[Stroustrup93] Stroustrup B. - *The C++ Programming Language*, 2nd Edition, Addison-Wesley Publishing Company, 691 pages. - 1993.

[Tailor76] Tailor R.W. et Frank R.L. - *CODASYL Database Management systems*, dans ACM Computing surveys, vol. 8(1), pp.67-103. - mars 1976.

[Tatsubori et al.98] Tatsubori M. et Chiba S. - *Yet another java.lang.Class*, dans Object-Oriented Technology, ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, Lecture Notes in Computer Science, vol. 153, pp. 372-373. - Bruxel, Belgique, juillet 1998.

[Tatsubori99] Tatsubori M. - *OpenJava Tutorial,* Université de Tsukuba, Japon, http ://www.hlla.is.tsukuba.ac.jp/~mich/openjava/ - 1999.

[Tsichritzis et al.76] Thischritzis D et Lochovsky F.H. - *Hierarchical Database Management : A Survey*, dans ACM Computing Surveys, vol. 8(1), pp. 105-123. - mars 1976.

[Thieme94] Thieme C. - *Integration of Database Systems Based on Structure and Behavior*, dans les actes de ERCIM'94, pp. 38-57. - 1994.

[Thomas et al.90] Thomas G., Thomson G.R., Chung C.W., Barkmeyer E., Carter F., Templeton M., Fox S. et Hartman B. - *Heterogeneous Database Systems for Production Use*, dans ACM Computing Surveys, vol. 22(3), pp. 237-266 - septembre 1990.

[Tomasic et al.95] Tomasic A., Rashid L. et Valduriez P., - *Scaling heterogenous Databases and the Design of DISCO*, rapport de recherche INRIA n°2704, 25 pages, - novembre 1995.

[Tresch et al.94] Tresch M. et Scholl M.H. - *A classification of MultiDatabase Languages*, dans les actes de 3rd International Conference on Parallel and Distributed Information systems (PDIS'94), pp. 195-202. - Austin, Texas, USA, septembre 1994.

[Uniface] Uniface, environnement de développement par composants, http ://www.compuware.com.

[Versant] Versant, système de gestion de bases de données à objets, www.versant.com.

[W3C99] W3C - Extensible Markup Language (XML) 1.0, W3C Recommendation, http ://www.w3.org/TR/1998/REC-xml-19980210 - février 1998.

[Widom95] Widom J. - *Research Problems in Data Warehousing*, dans les actes de 4th conference on information an Knowledge Management (CIKM'95), pp. 25-30. - Baltymore, Maryland, novembre 1995.

[Wing et al.92] Wing J.M., Faehndrich M., Morrisett J.G et Nettles S. - *Extension to Standard ML to Support Transactions*, dans les actes du ACM SIGPLAN Workshop on ML and its application, 15 pages - juin 1992.

[Wolski et al.90] Wolski A. et Veijalainen J. - *2PC Agent Method : Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase*, dans Databases : Theory, Design and Applications, publication basée sur les actes de 1st International Conference on Databases, parallel Architectures and their Applications (PARBASE-90) , ed. par Rishe N., Navathe S.B. et Tal D., pp. 268-287. - Miami Beach, Floride, USA, mars 1990

[Wu et al.97] Wu Z. et Schwiderski S. Reflective Java, *Making Java even more flexible*, rapport de recherche ANSA n° APM.1936.02, 20 pages - 1997.

[Zhu et al.96] Zhu Q. et Larson P.A. - *Building Regression Cost Models for Multidatabase Systems*, dans les actes de 4th International Conference on Parallel and Distributed Information Systems (PDIS'96), pp. 220-231. - Miami Beach, Floride, USA, décembre 1996.

Annexe A

Résumé du Formalisme

A.1 Base du Formalisme

A.1.1 Instanciation

symbole	signification
$\mathcal{O}$	ensemble des identifiants d'objet
O	sous-ensemble de $\mathcal{O}$
o_{Object}	racine d'héritage
o_{Class}	racine d'instanciation
γ	fonction d'instanciation
$\dashv$	relation d'instanciation
π	fonction de peuplement

- fonction d'instanciation : associe à chaque objet sa classe.
- fonction de peuplement : associe à chaque classe ou métaclasse l'ensemble de ses instances directes.

A.1.2 Héritage

symbole	signification
$\mathcal{M}_o$	ensemble des métaobjets
$\mathcal{M}_c$	ensemble des métaclasses
$\mathcal{C}$	ensemble des classes
β	fonction de sous-classement
$\prec$	relation d'héritage
ϖ	fonction d'héritage
$\widetilde{\pi}$	extension

- fonction de sous-classement : associe à chaque classe ou métaclasse l'ensemble de ses sous-classes.

- fonction de d'héritage : associe à chaque classe ou métaclasse l'ensemble de ses super-classes.

A.1.3 Types

symbole	signification
$o_{MetaType}$	racine d'instanciation des types
o_{Type}	racine d'héritage des types
$\mathcal{X}(\mathcal{T})$	ensemble des fonctions de combinaison sur $\mathcal{T}$
$\chi_{\mathcal{T}}$	fonction de combinaison sur $\mathcal{T}$
σ_i	fonction de type intrinsèque
σ_h	fonction de type hérité
σ	fonction de type (classe ou métaclasse)
σ_o	fonction de type (objet)
$\mathcal{T}$	ensemble des types

- fonction de combinaison sur $\mathcal{T}$: associe à chaque couple de type un nouveau type.
- fonction de type intrinsèque : associe à chaque classe ou métaclasse le type intrinsèque de ses instances directes.
- fonction de type hérité : associe à chaque classe ou métaclasse le type de ses instances directes, hérité de ses super-classes.
- fonction de type (classe ou métaclasse) : associe à chaque classe ou métaclasse le type de ses instances directes.
- fonction de type (objet) : associe à chaque objet son type.

A.1.4 Comportements

symbole	signification
$o_{MetaMethod}$	racine d'instanciation des méthodes
o_{Method}	racine d'héritage des méthodes
$\mathcal{X}(P^{fin}(\mathcal{B}))$	ensemble des fonctions de combinaison sur $P^{fin}(\mathcal{B})$
$\chi_{P^{fin}\mathcal{B}}$	fonction de combinaison sur $P^{fin}(\mathcal{B})$
μ_i	fonction de comportement intrinsèque
μ_h	fonction de comportement hérité
μ	fonction de comportement (métaobjets)
μ_o	fonction de comportement (objets)
$\mathcal{B}$	ensemble des méthodes

- fonction de combinaison sur $P^{fin}(\mathcal{B})$: associe à chaque couple d'ensembles de méthodes un nouvel ensemble de méthodes.
- fonction de comportement intrinsèque : associe à chaque classe ou métaclasse le comportement intrinsèque de ses instances directes.
- fonction de comportement hérité : associe à chaque classe ou métaclasse le comportement de ses instances directes, hérité de ses super-classes.

- fonction de comportement (classe ou métaclasse) : associe à chaque classe ou métaclasse le comportement de ses instances directes.
- fonction de comportement (objet) : associe à chaque objet son comportement

A.1.5 Quelques Objets

symbole	signification
o_{Atomic}	racine d'héritage des types atomiques
$o_{Boolean}$	classe des booléens
o_{Char}	classe des caractères
o_{Int}	classe des entiers
o_{Float}	classe des réels
o_{String}	classe des chaînes de caractères
$o_{Collection}$	racine d'instanciation des types de collection
o_{List}	racine d'instanciation des types de liste
o_{Bag}	racine d'instanciation des types de sac
o_{Set}	racine d'instanciation des types d'ensemble
o_{Tuple}	racine d'isntanciation des types de n-uplet
o_{Void}	objet vide
$o_{Accessor}$	racine d'héritage des accesseurs
$o_{ReadAccessor}$	classe des accesseurs en lecture
$o_{WriteAccessor}$	classe des accesseurs en écriture
$\mathcal{S}$	ensemble des chaines de caractères

A.1.6 Envoi de message

symbole	signification
sig	fonction de signature
ret	fonction de retour
$\ll$	relation de typage
$\mathcal{F}$	ensemble des fonctions d'association de $\mathcal{S}$ dans $\mathcal{B}$
α	fonction d'association de $\mathcal{S}$ dans $\mathcal{B}$
ρ	fonction de message
λ	fonction d'exécution
λ'	fonction d'envoi de message

- fonction de signature : associe à chaque méthode la signature de ses arguments
- fonction de retour : associe à chaque méthode le type de son résultat
- fonction d'association de $\mathcal{S}$ dans $\mathcal{B}$: associe à chaque classe ou métaclasse la méthode correspondant à chaque message
- fonction de message : associe à chaque classe ou métaclasse l'ensemble des messages que peuvent recevoir ses instances

A.2 Compatibilité de classes

A.2.1 Compatibilité de classes :

$$\forall o_1, o_2 \in \mathcal{M}_o, \mu_h(o_1) \cap \mu_i(o_2) \neq \emptyset \Rightarrow \rho(o_2) \subseteq \rho(o_1) \tag{A.1}$$

A.2.2 Compatiblité d'héritage

$$\forall o_1, o_2 \in \mathcal{M}_o, o_1 \prec o_2 \Rightarrow \rho(o_1) \supseteq \rho(o_2) \tag{A.2}$$

A.2.3 Compatibilité de métaclasses descendante

symbole	signification
$root$	fonction associant chaque métaclasse à sa racine d'héritage

- pour tout $mo_1, mo_2 \in \mathcal{M}_c$ tels que $\mu_i(mo_1) \cap \mu_h(mo_2) \neq \emptyset$, si $\rho(root(mo_1)) \subseteq \rho(root(mo_2))$ alors la compatibilité descendante est vérifiée.

A.2.4 Compatibilité de métaclasses ascendante

- Pour tout $o_1, o_2 \in \mathcal{M}_o$ tels que $\mu_i(o_1) \cap \mu_h(o_2) \neq \emptyset$, si $\rho(\gamma(o_1)) \subseteq \rho(\gamma(o_2))$ alors la compatibilité ascendante est vérifiée.

A.3 système transactionnel

A.3.1 Métaclasses de contrôle transactionnel

symbole	signification
$o_{TransactionAbleClass}$	racine d'instanciation des classes de contrôle transactionnel
$o_{TransactionAbleObject}$	racine d'héritage des classes de contrôle transactionnel
$\mathcal{M}_{ct}$	ensemble des classes de contrôle transactionnel
$\mathcal{O}_{ct}$	ensemble des objets de contrôle transactionnel
τ	fonction de contrôle transactionnel

- fonction de contrôle transactionnel : associe à chaque classe de contrôle transactionnel les méthodes pouvant faire l'objet d'un envoi de message transactionnel.

A.3.2 Transactions

symbole	signification
$o_{Transaction}$	racine d'héritage des classes de transaction

- Fonctions de contrôle de l'environnement transactionnel
  - *getBegin* : associe à chaque transaction la transaction dans laquelle elle commence.
  - *getAbort* : associe à chaque transaction la transaction dans laquelle elle annule.
  - *getCommit* : associe à chaque transaction la transaction dans laquelle elle valide.
  - *getSubs* : associe à chaque transaction l'ensemble de ses sous-transactions.
  - *getObjects* : associe à chaque transaction l'ensemble des objets transactionnels auxquels elle a accédé.
  - *waitForCommit* : associe à chaque transaction l'ensemble des transactions devant être terminées avant sa validation.
  - *waitForAbort* : associe à chaque transaction l'ensemble des transactions devant être terminées avant son annulation.
  - *abortPriority* : associe à chaque transaction sa priorité d'annulation
  - *preparePriority* : associe à chaque transaction sa priorité de préparation.
  - *commitPriority* : associe à chaque transaction sa priorité de validation.

A.3.3 Envoi de message transactionnel

symbole	signification
λ_t	fonction d'exécution transactionnelle
λ'_t	envoi de message transactionnel

A.3.4 Objets transactionnels

symbole	signification
$o_{TransactionalClass}$	racine d'instanciation des classes transactionnelles
$o_{TransactionalObject}$	racine d'héritage des classes transactionnelles
$\mathcal{M}_t$	ensemble des classes transactionnelles
O_t	ensemble des objets transactionnels

A.3.5 Comportements transactionnels

symbole	signification
$o_{BehaviorClass}$	racine d'instanciation des classes de comportement transactionnel
$o_{BehaviorObject}$	racine d'héritage des classes de comportement transactionnel
$\mathcal{M}_b$	ensemble des classes de comportement transactionnel
O_b	ensemble des comportements transactionnels
κ_s	fonction d'association transactionnelle structurelle
κ_t	fonction d'association transactionnelle

- fonction d'association transactionnelle structurelle : associe à chaque classe de comportement transactionnel une classe transactionnelle.
- fonction d'association transactionnelle : associe à chaque objet transactionnel un comportement transactionnel.

A.3.5.1 Atomicité

symbole	signification
$o_{AtomicBehavior}$	racine d'instanciation des classes de comportement atomique
$o_{AtomicObject}$	racine d'héritage des classes de comportement atomique
$\mathcal{M}_b$	ensemble des méthodes inversibles

- Fonctions utilisées lors de l'algorithme de gestion atomique :
  - inv : associe à chaque méthode inversible son inverse.
  - $trace$: associe à chaque transaction, la trace de son exécution sur chaque objet transactionnel à comportement atomique

A.3.5.2 Contrôle de concurrence par verrouillage

symbole	signification
$o_{LockingBehavior}$	racine d'instanciation des classes de comportement à verrouillage
$o_{LockingObject}$	racine d'héritage des classes de comportement à verrouillage

- Fonctions utilisées dans l'algorithme de contrôle de concurrence
  - $rConcurrentWith$: associe à chaque transaction t et pour chaque objet transactionnel o l'ensemble des transactions en concurrence avec t pour lire o.
  - $wConcurrentWith$: associe à chaque transaction t et pour chaque objet transactionnel o l'ensemble des transactions en concurrence avec t pour écrire o.
  - $concurrentWith$: associe à chaque transaction t et pour chaque objet transactionnel o l'ensemble des transactions en concurrence avec t sur o.
  - $rLock$: associe à chaque objet transactionnel o, l'ensemble des transactions ayant un verrou en lecture sur o.
  - $irLock$: associe à chaque objet transactionnel o, l'ensemble des transactions ayant un verrou d'héritage en lecture sur o.

- *iwLock* : associe à chaque objet transactionnel o, l'ensemble des transactions ayant un verrou d'héritage en écriture sur o.
- *wLock* : associe à chaque objet transactionnel o, la transaction ayant un verrou en écriture sur o.

A.4 relais et requêtes

A.4.1 requêtes

symbole	signification
o_{Query}	racine d'héritage des classes de requêtes
where	fonction de restriction
proj	fonction de projection

- fonction de restriction : associe à chaque requête sa clause de restriction
- fonction de projection : associe à chaque requête sa clause de projection

A.4.2 ensembles résultats de requêtes

symbole	signification
o_{qSet}	racine d'instanciation des ensembles résultats de requêtes (e.r.r)
$o_{qSetOfObject}$	classe des e.r.r. contenant des o_{Object}
query	associe à chaque e.r.r. une requête
elements	associe à chaque e.r.r. l'ensemble de ses éléments
resultElements	associe à chaque e.r.r. l'ensemble résultat

A.4.3 Métaclasses de requêtes

symbole	signification
$o_{QueryAbleClass}$	racine d'instanciation des classes pouvant faire l'objet de requêtes
$o_{QueryAbleObject}$	racine d'héritage des classes pouvant faire l'objet de requêtes

A.4.4 Relais et sources de données

symbole	signification
$o_{ProxyClass}$	racine d'instanciation des classes de relais
$o_{ProxyObject}$	racine d'héritage des classes de relais
$o_{DataSources}$	racine d'héritage des classes de sources de données
$\mathcal{C}_r$	ensemble des classes de relais
O_r	ensemble des relais

- Fonctions d'identifications
  - *base* : associe à chaque classe de relais la base depuis laquelle sont issues les objets qu'elle représente.
  - *name* : associe à chaque classe de relais le nom de l'entité qu'elle represente.
  - id_c : associe à chaque classe l'expression identifiant de manière unique les objets qu'elle représente.
  - id_o : associe un identifiant à chaque relai.
- Fonctions de contenu
  - *data* : associe des données à chaque relai.
  - *max* : associe à chaque classe de relais le nombre maximum de relais contenant des données en mémoire.
  - *min* : associe à chaque classe de relais le nombre minimum de relais à décharger avant le chargement de nouveaux relais.
  - *isUnloadable* : fonction associant à chaque relai un booléen indiquant sa capacité à être déchargé.
  - *memObjects* : fonction associant à chaque classe de relais l'ensemble des relais chargés.
  - *deletedObjects* : fonction associant à chaque classe de relais l'ensemble des objets devant être détruits et rendus non persistants.
  - *newObjects* : fonction associant à chaque classe de relais l'ensemble des objets devant être rendus persistants.

A.5 vues

A.5.1 Classes virtuelles

symbole	signification
$o_{VirtualClass}$	racine d'instanciation des classes virtuelles
$o_{VirtualObject}$	racine d'héritage des classes virtuelles
$\mathcal{C}_v$	ensemble des classes virtuelles

A.5.2 Mappings et objets virtuels

symbole	signification
$o_{MappingClass}$	racine d'instanciation des mappings
$o_{MappingObject}$	racine d'héritage des mappings
$\mathcal{C}_m$	ensemble des mappings
O_v	ensemble des objets virtuels

A.5.3 Structuration et mapping

symbole	signification
$struct$	fonction de structuration
map	fonction de mapping
$\widetilde{\pi}_v$	extension virtuelle
$base_m$	fonction de base de mapping
$base_v$	fonction de base virtuelle

- fonction de structuration : associe à chaque mapping la classe virtuelle qui lui correspond.
- fonction de mapping : associe à chaque classe virtuelle l'ensemble des mappings actifs qui lui correspondent.
- fonction de base de mapping : associe à chaque mapping l'ensemble des classes de base qui lui correspondent.
- fonction de base virtuelle : associe à chaque objet virtuel les instances de base qui lui correspondent.

A.5.4 Correspondances

symbole	signification
$o_{Mapping}$	racine d'héritage des classes de correspondances
$o_{SimpleMapping}$	classe des correspondances simples
$o_{ComplexMapping}$	classe des correspondances complexes
Ξ	ensemble des correspondances
$corr$	fonction de correspondance
$constraint_s$	fonction de contrainte simple
$constraint_c$	fonction de contrainte complexe

- fonction de correspondance : associe à chaque mapping l'ensemble des correspondances qui le compose.
- fonction de contrainte simple : associe à chaque mapping une contrainte simple.
- fonction de contrainte complexe : associe à chaque mapping une contrainte complexe.
- fonctions utilisées dans les algorithmes de gestion des vues

A.5.5 Vues et héritage

Si l'on souhaite que la sémantique du lien d'héritage soit vérifiée pour les classes virtuelles, il est nécessaire que :

- $\forall c, c' \in \widetilde{\pi}(o_{VirtualClass})$, si $c \prec c'$ alors $\forall m' \in map(c')$ il existe $m \in map(c)$ tel que $m' \prec m$.