N° d'ordre :

THÈSE

présentée à

L'UNIVERSITÉ PARIS XI

pour obtenir le grade

DOCTEUR DE L'UNIVERSITÉ PARIS XI

Spécialité

INFORMATIQUE

par

Jérôme SIMÉON

Titre :

Intégration de sources de données hétérogènes

Ou comment marier simplicité et efficacité

Soutenue le 18 janvier 1999 devant le jury composé de :

MM.	Claude DELOBEL	Directeur
	Georges GARDARIN	Rapporteurs
	Richard HULL	
Mme	Sophie CLUET	Examinatrice
MM.	Serge ABITEBOUL	Examinateurs
	Jean-Pierre JOUANNAUD	

Je remercie Sophie Cluet qui m'a guidé durant ces trois années.

Merci pour tout ce que tu m'as appris, pour la grande liberté dont j'ai bénéficié, pour tout le plaisir que j'ai eu à travailler avec toi. Merci pour m'avoir aidé à garder le cap (parfois même remis dans le droit chemin !) et pour avoir entretenu le moral des troupes. Merci enfin, Sophie, pour ton amitié.

Je remercie Claude Delobel qui a accepté de diriger ma thèse.

Merci pour les conseils éclairés et les discussions enrichissantes. Merci d'avoir été présent chaque fois que j'en ai eu besoin.

Je remercie Serge Abiteboul qui me fait le grand plaisir de participer à mon jury.

Merci de m'avoir accueilli à Verso, je m'y suis senti chez moi. Et merci pour m'avoir aidé quand j'ai voulu en partir !

Je remercie Georges Gardarin et Richard Hull pour avoir accepté de rapporter sur ma thèse et pour les judicieuses remarques qu'ils m'ont faites. Je remercie Jean-Pierre Jouannaud qui me fait l'honneur de présider ce jury et à qui j'ai eu le plaisir de présenter mon travail lors d'une trop courte après-midi.

J'aimerais adresser un remerciement tout particulier à Patrick Duval. Bien que je n'en aie pas eu conscience à l'époque, c'est lui qui, le premier, m'a fait découvrir et aimer la recherche.

Cher Patrick, je ne pourrai jamais assez te remercier.

Je remercie Vassilis Christophides qui m'a aidé a faire mes premiers pas dans l'univers des bases de données, et bien plus encore.

Merci pour avoir cru en moi dès le début. Merci pour cet enthousiasme communicatif qui m'a entraîné jusqu'en Grèce. Merci d'être toi.

Tout au long de ma thèse, j'ai bénéficié d'un environnement et de contacts exceptionnels. Je remercie tous ceux avec qui j'ai eu la chance de discuter de mon travail et qui m'ont offert idées et conseils.

Merci à Emmanuel Waller qui a écouté mes élucubrations. Merci à Didier Rémy et Pierre Weis qui ont bien voulu s'intéresser un instant aux tableaux à deux dimensions. Merci à Anne-Marie Vercoustre, Daniel Chan et Rodney Topor pour leurs très pertinentes suggestions.

Je remercie les membres de l'équipe Verso, anciens et nouveaux, pour l'ambiance et la bonne humeur. Vous me manquerez tous beaucoup.

Merci à Bernd, Cássio, Fanny, Luc, Leonid, Philippe, Pierangelo, Stéphane, Yoshito et Zoé. Merci à Irini et Laurent qui ont relu de nombreuses parties de ce manuscrit. Merci à Michel Scholl pour sa profonde gentillesse et sa lecture attentive de mon introduction.

Je remercie Danny pour son soutien au fil des jours.

Merci d'avoir relu toute ma thèse. Merci de t'être toujours occupée de tout.

Je remercie mes parents, que je n'ai pas beaucoup vu ces derniers temps, mais qui sont restés près de moi.

Moi non plus, Sihem, je ne trouve pas de mots pour te remercier.

Sommaire

Chapitre 1

Introduction

I think there is a world market for maybe five computers.
– Thomas Watson, Chairman of IBM, 1943

Si l'informatique était, hier encore, principalement utilisée pour le calcul numérique, elle intervient désormais dans tous les domaines. Elle est devenue ces dernières années un outil incontournable et c'est à juste titre que l'on peut parler de « révolution ». La démocratisation de l'ordinateur auprès du grand public et sa généralisation dans les entreprises ont suscité peu à peu un recours croissant à l'informatique. Mais ce n'est que récemment que le phénomène s'est emballé, sous l'impulsion transmise par l'apparition de la *Toile*. Cette agora virtuelle, tissée au-dessus du réseau mondial internet, a eu pour effet de simplifier de façon radicale l'échange des données numériques. Pour profiter de cette capacité à communiquer sans précédent, c'est l'intégralité des informations nécessaires à l'activité humaine qui est en passe d'être stockée sous forme numérique. Tout naturellement, l'envie d'exploiter ces données s'en trouve également décuplée. Une même information doit ainsi pouvoir parvenir à de multiples catégories d'utilisateurs, ayant chacune des besoins différents. Pour fonctionner, les applications doivent donc désormais accéder à des sources de données de plus en plus nombreuses et variées. Ce nouveau contexte pose de sérieux problèmes aux systèmes de gestion de bases de données : c'est le point de départ du travail que nous présentons ici.

Lorsqu'une application souhaite pouvoir manipuler les données de sources distantes et hétérogènes, les difficultés s'accumulent rapidement. Ces difficultés, qui découlent des nombreuses différences entre les sources, sont malheureusement inévitables. Tout d'abord, les formes de représentation des données peuvent être extrêmement variées. Elles dépendent la plupart du temps de l'application initiale (p.ex. Word™ ou SGML pour la réalisation de documents, tables relationnelles pour les applications de gestion) ou de l'outil qui a servi à les créer. Cette représentation convient donc rarement à l'application finale. Ensuite, les sources peuvent être différentes dans leurs formats de stockage, dans la sémantique des données ou dans les opérations disponibles sur ces données. Pour couronner le tout, chacune d'elles doit être manipulée à travers un protocole et une interface bien précis. Ainsi, l'accès aux informations, qui restait relativement simple dans le cas d'une base de données centralisée, se complique rapidement lorsque les sources se multiplient. Ces problèmes d'hétérogénéité, ainsi que le volume parfois important des données à traiter, font du développement de telles applications un véritable casse-tête. Faciliter la réutilisation de sources de données hétérogènes est devenu un enjeu majeur.

Il existe des solutions pour de nombreux cas particuliers, incluant des conversions entre formats (LGrind [LGr98], HEVEA [Mar98], O_2Web [O2W96]), des familles de transformations dans un même format (DSSSL [ISO96] pour les documents structurés, EXPRESS-M [Bai95] pour les données techniques ou encore les langages de vues des bases de données [Sou95]) et certains types d'intégration plus traditionnels (fédération de bases de données relationnelles [SL90]). Cependant, les outils proposés sont spécifiques aux types de sources pour lesquelles ils ont été conçus et souvent difficiles à paramétrer, ce qui limite la possibilité de les adapter aux besoins d'une application particulière. Dans la pratique, il est rare de pouvoir tirer parti du travail accompli pour leurs réalisations : un contexte applicatif parfois très proche (procéder à une conversion impliquant deux documents, rajouter une source non relationnelle dans une base de données fédérée) nécessite encore trop souvent de repartir de zéro.

L'objectif des *systèmes d'intégration* est de définir un cadre général pour faciliter la réutilisation de sources de données hétérogènes. Ils adoptent pour cela une méthode naturelle qui consiste à traduire l'ensemble des informations dans une langue commune. Cette phase d'intégration, destinée à donner l'impression que l'on accède à une source unique, précède le développement de l'application proprement dite. Le système d'intégration doit, quant à lui, fournir les outils nécessaires : dans l'idéal, le développeur ne devrait avoir à s'occuper que des aspects sémantiques, dont le traitement ne peut être complètement automatisé. Jusqu'à présent, deux approches s'opposent. Les systèmes dits *structurés* utilisent les technologies des bases de données, ils offrent de bonnes performances mais l'intégration nécessite un travail considérable. Les systèmes dits *semistructurés* utilisent de nouveaux modèles pour la représentation des données, ils permettent de simplifier l'intégration, mais leurs performances laissent encore à désirer.

Ce travail porte sur la conception et l'implantation d'un ensemble d'outils destinés à faciliter la réutilisation de l'information présente sous forme numérique. Nous défendons la thèse qu'il est possible de mettre au point un système qui minimise la quantité de travail lors de la mise en place de l'intégration et qui soit également performant lors de son utilisation.

Systèmes d'intégration : de la simplicité...

Pour permettre la manipulation de données provenant de sources hétérogènes, tous les systèmes d'intégration sont conçus autour d'une représentation pivot et d'une architecture avec deux types de composants. Les *adaptateurs* exportent les informations de chaque source dans la représentation pivot, puis le *médiateur* se charge du reste de l'intégration dans l'environnement homogène fourni par cette représentation. Dans un premier temps, nous nous intéressons au travail nécessaire pour le développement et la mise en place de ces composants.

Bâtis sur les modèles relationnel (comme DISCO [TRV96]) ou orienté-objet (comme Garlic [CHS$^+$95] ou IRO-DB [GGF$^+$95]), les systèmes structurés imposent aux sources d'exporter leurs informations par rapport à un schéma d'intégration global. La résolution des conflits sémantiques entre les sources et ce schéma est donc à la charge des adaptateurs. Ceux-ci sont alors réalisés pour une seule application et nécessitent un travail important – travail remis en cause chaque fois qu'un changement doit s'opérer (p.ex. ajout d'une source, modification du schéma de l'application).

Inversement, certains systèmes (comme TSIMMIS [PGW95]) choisissent un modèle pivot semistructuré afin d'optimiser la mise en place de l'intégration. Dans ce modèle les informations sont représentées à l'aide d'arbres annotés et aucun schéma préalable n'est requis, ce qui offre une grande flexibilité. Plus important dans le contexte d'intégration qui nous occupe, un tel modèle permet de représenter *naturellement* les données provenant de sources quelconques – en s'inspirant par exemple de leur arbre de syntaxe. Cette dernière propriété est fondamentale car elle permet la conception d'adaptateurs *génériques* pour des familles de sources (p.ex. un adaptateur pour les documents structurés, un adaptateur pour les bases de données relationnelles). Le travail accompli pour l'implantation des adaptateurs est ainsi largement réutilisable et ceux-ci peuvent être fournis avec le système d'intégration. Enfin, conjugué à la flexibilité du modèle semistructuré, cela permet une adaptation rapide aux changements qui s'opèrent au niveau des sources.

Les adaptateurs ne procèdent à aucune interprétation des données lors de leur conversion vers le format pivot, la résolution de l'hétérogénéité sémantique est donc à la charge du médiateur. Celui-ci doit, en effet, pouvoir fournir à l'application finale une vue cohérente et homogène de l'ensemble des informations. Pour cela, les systèmes d'intégration semistructurés fournissent des langages de haut niveau (inspirés des langages de requêtes pour bases de données) permettant une spécification déclarative du travail de restructuration que doit fournir le médiateur. Un tel langage simplifie beaucoup la réalisation du médiateur, malheureusement l'utilisation du modèle semistructuré devient alors un handicap. L'absence de schéma rend en effet difficile la compréhension rapide de la structure et de la sémantique des données, nécessaire à l'écriture du programme d'intégration. De plus, il n'est pas possible d'en vérifier la validité, opération habituellement effectuée par rapport au type des données.

Dans la première partie de cette thèse, nous proposons un ensemble d'outils destinés à simplifier la conception des médiateurs dans les systèmes d'intégration semistructurés. Plus précisément, nous introduisons un système de types adapté, un langage déclaratif pour la spécification de programmes d'intégration et des mécanismes pour faciliter l'écriture de ces programmes. Enfin, nous présentons le *système YAT* [1], un prototype de système d'intégration semistructuré qui valide l'approche proposée.

Typage d'un modèle semistructuré. Nous étendons le modèle de données semistructuré par un système de types. Fondé sur la notion d'*instanciation*, celui-ci autorise la description d'informations structurelles à différents niveaux d'abstraction, du plus générique (pour plus de souplesse) au plus spécifique (pour plus de contrôle). La précision du typage obtenu permet d'adapter celui-ci à la nature des données traitées et à l'application envisagée, tout en conservant la flexibilité initiale du modèle semistructuré. Le typage et le mécanisme d'instanciation sous-jacent joueront un rôle clé dans l'ensemble des techniques présentées tout au long de cette dissertation.

Le système de types est utile en premier lieu pour décrire la structure des données exportées par les adaptateurs. Dans ce contexte, nous étudions sa capacité à représenter exactement les types d'origine des sources (p.ex. le schéma d'une base de données, la grammaire d'une famille de documents) ce qui, lorsque c'est le cas, fournit l'information structurelle la plus précise possible. Enfin, nous démontrons la faculté du modèle à s'adapter à une connaissance imparfaite de la structure des données, propriété indispensable pour le traitement de certains formats de documents (comme XML ou encore BibTeX).

Un langage d'intégration déclaratif. YAT$_L$ est un langage à base de règles conçu pour la spécification déclarative de l'intégration réalisée par le médiateur. Il s'apparente à un lan-

1. `http://www-rocq.inria.fr/~simeon/YAT/`

gage de vues pour données semistructurées et propose, notamment : des filtres autorisant un traitement concis et intuitif des données sources, des opérations pour contrôler la création de nouvelles données et le partage d'information, des primitives de haut niveau pour la manipulation des collections et la possibilité d'introduire de nouvelles informations grâce à des fonctions externes.

Nous montrons comment YAT$_L$ peut être utilisé pour résoudre d'éventuels conflits sémantiques et créer une représentation homogène à partir des données exportées par les adaptateurs d'un système d'intégration. Par ailleurs, nous illustrons la capacité du langage à décrire des programmes d'intégration génériques, c'est-à-dire fonctionnant sur de larges familles de sources. Ce type de programme est particulièrement utile lorsque certaines de ces sources évoluent rapidement ou dans le cadre d'applications de conversion (par exemple, offrir la vue d'une base de données à objets sur la Toile nécessite de pouvoir transformer toute valeur ODMG en page HTML).

YAT$_L$ est compatible avec le système de types. Les informations structurelles fournies par les adaptateurs s'avèrent particulièrement utiles pour guider l'écriture des programmes d'intégration. Il est possible de procéder à des vérifications sur les types d'entrée et de sortie des programmes afin de détecter d'éventuelles erreurs. Ces vérifications sont nécessaires dans les applications de conversion pour assurer la conformité des données créées par rapport à un format de sortie.

Instanciation automatique de programmes. Nous avons été amenés à définir des programmes d'intégration plus ou moins génériques. Nous montrons comment la notion d'instanciation s'étend aux programmes YAT$_L$. Nous en déduisons plusieurs mécanismes d'ordre syntaxique pour dériver automatiquement des programmes spécifiques à partir de programmes génériques. Il est possible, par exemple, d'instancier un programme générique du modèle ODMG vers le format HTML, en un programme ayant une sémantique équivalente mais fonctionnant sur le schéma d'une base de données particulière. D'un point de vue pratique, ces mécanismes donnent au développeur la possibilité d'adapter un programme existant à ses besoins plutôt que de partir de zéro. Nous nous intéressons à la cohabitation entre transformations génériques et spécifiques, et introduisons des mécanismes de contrôle pour assurer la résolution des conflits qui peuvent survenir.

Le système YAT. Nous décrivons l'architecture et l'implantation d'un système d'intégration semistructuré qui fournit l'ensemble des outils décrits ici. Le prototype comprend principalement un médiateur, conçu autour d'un interprète pour le langage YAT$_L$, et des adaptateurs pour quelques familles de sources courantes (notamment les documents HTML et BibTeX, et la base de données à objets O_2). L'architecture des adaptateurs est conçue pour assurer une réutilisation optimale des modules qui les constituent. Le système propose en standard des composants applicatifs d'intérêt général, notamment une interface de requête générique simple, et des composants, dits « adaptateurs de sortie », permettant d'exporter les données générées vers un format cible dans le cadre d'applications de conversion. Ce prototype a été validé sur plusieurs applications, dont certaines réalisées par nos partenaires industriels dans le cadre du projet européen OPAL[2]. L'utilisation du système pour l'intégration (notamment la création de sites sur la Toile à partir de sources hétérogènes) ou la conversion entre formats,

2. *Integrated Information and Process Management In Manufacturing Engineering*, Projet Esprit IV N° 20377. `http://www.cordis.lu/esprit/src/20377.htm`

démontre sa capacité à servir de support au développement rapide d'applications impliquant la réutilisation de sources de données hétérogènes.

... à l'efficacité

Nous avons désormais un ensemble d'outils permettant la spécification et la mise en place rapide de l'intégration. Dans la deuxième partie de cette thèse, nous nous intéressons aux problèmes d'efficacité lors de l'évaluation de requêtes sur les données exportées par un système d'intégration. Il s'agit là d'une préoccupation importante dans la mesure où certaines applications impliquent la manipulation de gros volumes de données.

Afin de permettre le stockage et l'accès à de grandes quantités d'information, les systèmes de gestion de bases de données proposent des techniques d'optimisation appropriées. Celles-ci sont souvent fondées sur un modèle d'évaluation algébrique et sur un optimiseur capable de tirer parti des équivalences de l'algèbre choisie (relationnelle ou orienté-objet) pour trouver une évaluation efficace. Notons que, lorsque les données sont distribuées, être capable de répartir l'évaluation de manière optimale sur les différents sites est une technique décisive [OV91].

Construits sur les modèles des bases de données, les systèmes structurés incorporent ces techniques dans leurs médiateurs. Qui plus est, certains d'entre eux [HKWY97, KTV97] autorisent les sources à déclarer les opérations qu'elles sont capables d'évaluer, dans le but de soulager le médiateur d'une partie du travail. C'est donc fort logiquement qu'ils offrent d'excellentes performances. Malheureusement, en informant le médiateur des capacités d'une source, un adaptateur doit également être capable de transformer une opération relative au schéma d'intégration en une opération spécifique à la source correspondante. Une fois de plus, ceci peut nécessiter un travail de développement considérable.

D'un autre côté, l'accès aux données dans les systèmes semistructurés reste coûteux dans bon nombre de cas. En l'absence d'une algèbre adaptée, les techniques d'optimisation mises au point pour ces systèmes restent essentiellement d'ordre syntaxique ce qui les rend difficiles à utiliser dans le cas général. En l'absence d'information de types, les réécritures utilisées ne peuvent exploiter que les informations structurelles implicites dans les programmes d'intégration ce qui, une fois de plus, limite leur portée. La description des capacités opérationnelles d'une source, lorsqu'elle est possible [PGGU95], est de nouveau spécifique au schéma. Cela est incompatible avec le caractère générique des adaptateurs, dont on a vu qu'il simplifiait considérablement la phase d'intégration. Enfin, la qualité de la description ne permet généralement pas de tirer parti de sources non conventionnelles (tels que les systèmes de recherches d'information).

L'existence d'un système de types pour notre modèle pivot ainsi que l'utilisation d'un langage d'intégration déclaratif nous donnent l'opportunité de mettre au point des techniques d'optimisation appropriées. Dans le but de réconcilier la simplicité d'utilisation des systèmes semistructurés avec une évaluation efficace, nous introduisons un modèle opérationnel adapté. Nous montrons comment ce modèle autorise une description générique des capacités de sources hétérogènes et permet d'élaborer des techniques d'optimisation dans un contexte distribué.

Un modèle opérationnel pour les systèmes d'intégration semistructurés. Nous proposons un modèle opérationnel pour le système YAT. Celui-ci est conçu autour d'une algèbre pour données semistructurées capable de décrire l'évaluation de programmes YAT$_L$ ainsi que l'évaluation de requêtes formulées par l'application finale. Nous choisissons l'algèbre objet de Cluet et Moerkotte [CM93] à laquelle nous ajoutons deux nouveaux opérateurs pour la manipulation des données semistructurées. Nous introduisons de nouvelles équivalences impliquant les opérateurs semistructurés qui, conjuguées aux équivalences algébriques usuelles, permettent de retrouver la majorité des optimisations classiques utilisées dans les bases de données centralisées.

Exploiter les capacités des sources. En nous fondant sur ce modèle nous exploitons, une fois de plus, les fonctionnalités de typage disponibles pour permettre une description complète des capacités opérationnelles des sources. Nous étendons la notion d'instanciation sur les opérateurs de l'algèbre, ce qui donne la possibilité de décrire pour chaque source les opérations qu'elle supporte et sur quels types de données. La qualité du typage permet de représenter des capacités génériques (par exemple tout OQL pour n'importe quelle base de données ODMG), ce qui permet de préserver la généricité des adaptateurs. Le modèle opérationnel est facilement extensible par de nouvelles opérations, ou par de nouvelles équivalences algébriques. Cette extensibilité permet de décrire et d'exploiter les capacités exotiques offertes par certaines sources.

De l'ensemble des outils élaborés dans ce travail, il résulte une connaissance précise des sources tant du point de vue des structures de données que des opérations disponibles. Pour finir, nous montrons que cette connaissance permet non seulement de retrouver les techniques d'optimisation distribuées, mais également de concevoir des optimisations originales, impliquant, par exemple, une connaissance partielle de la structure des données ou l'utilisation de sources dont les capacités sont atypiques.

Plan

Avant d'entrer dans le vif du sujet, nous revenons dans le chapitre 2 sur l'ensemble des problèmes posés par l'intégration de sources de données hétérogènes. Nous en profitons pour introduire l'approche que nous préconisons, l'architecture adaptateur/médiateur, ainsi que les différents éléments qui seront étudiés par la suite. Le chapitre 3 est une étude détaillée des travaux liés à la réutilisation des données, ce qui clôt l'entrée en matière. Le chapitre 4 présente le modèle de données YAT et son système de types. Le chapitre 5 décrit YAT_L, ses interactions avec le système de types et les mécanismes de contrôle qui s'y rattachent. Le chapitre 6 introduit le modèle opérationnel. Nous montrons comment ce modèle peut servir à décrire les capacités de sources hétérogènes et étudions les possibilités optimisation qu'il procure. Finalement, le chapitre 7 s'intéresse au système YAT proprement dit. Nous décrivons son architecture, donnons quelques détails sur son implantation, puis illustrons son utilisation à l'aide de quelques applications.

Chapitre 2

Problématique et approche

At the moment it's just a Notion, but with a bit of backing I think I could turn it into a Concept, and then an Idea.

– Woody Allen
(in *Annie Hall*, 1977)

Sans revenir sur les raisons qui nous poussent à nous intéresser à la réutilisation de sources de données hétérogènes, rappelons que notre objectif est de fournir les outils permettant de faciliter cette opération. L'ambition de ce chapitre est double. Tout d'abord, il s'agit d'introduire l'intégration de sources de données hétérogènes en tant que sujet d'étude à part entière et d'illustrer la diversité et la complexité des problèmes que celui-ci met en jeu. Ensuite, il s'agit d'offrir un cadre général au développement de cette thèse. En situant le travail réalisé dans son contexte, nous espérons ainsi permettre au lecteur de mieux en distinguer la portée et les limites.

Pour cela, nous commençons (section 2.1) par préciser ce que nous entendons par intégration de sources d'information. Nous décrivons ensuite (section 2.2) les éléments qui composent une source et expliquons en quoi leur hétérogénéité, et particulièrement (section 2.3) celle des formats de représentation qu'elles utilisent, va nous causer de sérieuses difficultés. Cela nous permettra notamment (section 2.4) de justifier l'architecture et l'approche que nous avons adoptées.

2.1 Qu'est-ce que l'intégration ?

La figure 2.1 illustre la problématique de l'intégration dans le cas général. Une application a besoin pour son fonctionnement d'une certaine information dans un format précis (**donnée4** au format4). Cette information peut être obtenue à partir d'autres, fournies par des sources déjà en place mais dans des formats distincts (**donnée1** au format1, **donnée2** au format2, **donnée3** au format3). Afin de réaliser cette opération, un composant logiciel – que nous appellerons *module d'intégration* – est nécessaire.

Il ne s'agit certainement pas d'un problème nouveau et de nombreux travaux antérieurs ont porté sur des cas particuliers ou des classes de problèmes similaires. Avant de rentrer dans le vif du sujet, nous souhaitons préciser quelques points de vocabulaire et évoquer certains de

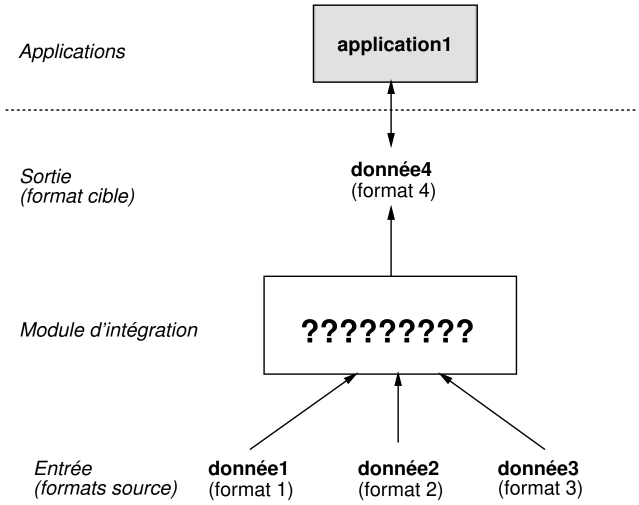


FIG. 2.1: *Réutiliser les informations existantes*

ces travaux. Dans un premier temps, quelques critères simples suffiront pour les distinguer :

1° Les données de sortie sont-elles *virtuelles*, ou *matérialisées* ? Dans le premier cas, les données requises par l'application (et seulement celles-ci), sont converties au fur et à mesure des besoins. Dans le second cas, une première phase consiste à convertir l'ensemble des données vers le format cible, avant que l'application ne puisse les utiliser.

2° Doit-on maintenir la *correspondance* qui existe entre données d'entrée et de sortie ? Si oui, il faut fournir les mécanismes permettant de propager une mise à jour effectuée des sources vers la cible et *vice versa*.

Certaines caractéristiques moins importantes peuvent rentrer en ligne de compte comme, par exemple, le nombre de sources. En fonction de ces différents critères, voici la terminologie que nous utiliserons par la suite[1] :

Intégration. Nous parlerons d'intégration de données pour désigner l'ensemble des problèmes tels que décrits sur la figure 2.1.

Conversion. Nous parlerons de conversion de données lorsque toutes les données sont matérialisées et lorsqu'il n'est pas nécessaire de maintenir le lien entre sources et cible. On peut alors imaginer que les données d'origine sont jetées. Dans de nombreux cas, la conversion n'implique qu'un seul format source. Les termes de *migration* ou encore de *chargement en masse* sont utilisés pour indiquer une conversion impliquant de gros volumes de données.

Vues. Nous parlerons de vues dans le cas où la correspondance entre données d'entrée et données de sortie doit être maintenue. Une vue peut être matérialisée ou virtuelle, voire partiellement matérialisée. Le terme d'*entrepôt de données* est employé dans certaines classes d'applications utilisant des vues matérialisées.

1. NDLR : le lecteur familier des bases de données retrouvera ici un vocabulaire connu. Il ne s'agit cependant pas d'une terminologie officielle ni acceptée par tous, chacun de ces termes étant utilisé parfois dans des contextes très différents, voire de façon contradictoire. L'objectif est de distinguer certains cas très largement étudiés dans la littérature et de situer ceux-ci par rapport au problème étudié ici.

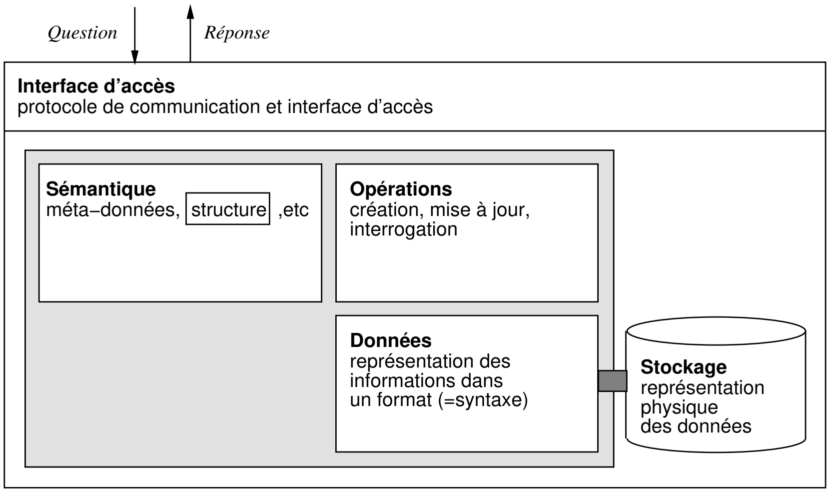


Fig. 2.2: *Schéma d'une source d'information*

Bien que nous traitions d'intégration dans son sens le plus général, il nous arrivera de revenir sur les spécificités liées à l'un des sous-problèmes ci-dessus. Par ailleurs, le chapitre 3 nous donnera l'occasion d'évoquer les travaux antérieurs dans chacun de ces domaines.

2.2 Nos sources sont hétérogènes !

L'intégration de sources de données est rendue délicate en raison de leur *hétérogénéité*. Cette notion couvre, en fait, un ensemble de problèmes très divers. L'objectif de cette section est de fournir un aperçu de ces problèmes et de distinguer clairement ceux d'entre eux que nous traiterons par la suite.

Qu'est-ce qu'une source d'information ? Dans l'absolu, toute source doit permettre la représentation, le stockage et l'exploitation ultérieure d'un ensemble d'informations. Pour cette raison, elles fournissent toutes un certain nombre de services communs. La figure 2.2 propose la vision schématique d'une source d'information et de ces « services minimum ».

- **Données :** Elles constituent le support informatique permettant de représenter l'information. Elles sont conservées dans un certain *format*, c'est à dire écrites en respectant une certaine syntaxe. Pour des raisons de performance, le stockage physique des données peut utiliser une représentation différente. Celle-ci étant habituellement invisible à l'utilisateur, nous ne nous en préoccuperons pas dans la suite.

- **Opérations :** Chaque source peut effectuer des opérations sur les données qu'elle stocke, permettant ainsi la création, l'extraction ou la modification d'informations.

- **Sémantique :** Retrouver l'information d'origine à partir des données brutes nécessite une phase d'*interprétation* qui peut s'avérer fort complexe. Chaque source fournit donc des informations additionnelles (ou informations *sémantiques*) afin de faciliter la compréhension des données stockées.

La *structure* (ou encore le *schéma* ou le *type*) est une catégorie d'information additionnelle particulièrement importante car elle fait le lien entre la sémantique des données et leur syntaxe. Elle joue de ce fait un rôle central dans l'architecture des sources d'information et jouera un rôle tout aussi central dans la suite de ce travail.

- **Interface d'accès :** Enfin, chaque source offre une couche logicielle permettant l'utilisation des services que nous venons de décrire.

Malheureusement, chacun de ces services fait l'objet de choix différents d'une source à l'autre et induit par la même occasion de l'hétérogénéité. Nous passons maintenant en revue les différents choix possibles pour chacune de ces catégories ainsi que les problèmes d'hétérogénéité qui en résultent : hétérogénéité des interfaces d'accès (ou *interopérabilité*), hétérogénéité des opérations, hétérogénéité des formats de représentation des données, et pour finir hétérogénéité sémantique.

Interopérabilité. Les données sont gérées par des systèmes extrêmement divers, offrant des moyens d'accès et de communication différents. Il peut s'agir de systèmes de gestion de bases de données, de systèmes de recherche d'information, de serveurs sur la Toile, de systèmes de fichiers, etc. Chacun de ces systèmes peut être manipulé directement (fichiers), à travers un protocole réseau de plus ou moins haut niveau (ftp [PR85], http [FGM$^+$97], CORBA [COR98]), à travers une interface de programmation (ODBC [ODB94] pour les bases de données, SDAI [ISO94b] pour les données techniques), voire une interface utilisateur (formulaires HTML).

Il faut être capable d'accéder aux données par l'un quelconque de ces moyens. Faute de pouvoir échapper à la réalisation de modules permettant l'accès à travers toutes ces interfaces, il est indispensable d'offrir une architecture suffisamment souple pour ne pas écrire plusieurs fois tout ou partie de ces modules.

Hétérogénéité des opérations. Chacun de ces systèmes propose des primitives différentes pour interroger et modifier les données. L'interrogation peut être faite par navigation (systèmes de fichiers, hypertextes, bases de données à objets), par mots-clés, requêtes booléennes (systèmes de recherches d'information [BC92, LH94]), expressions régulières (grep [Ste94] pour les fichiers), ou encore en utilisant un langage de requête déclaratif (SQL [Mel88, MS93] ou OQL [Clu91, Cat97, Clu98] dans les systèmes de gestion de bases de données).

Dans un premier temps, il est indispensable de pouvoir s'adapter à ces différentes capacités pour accéder aux informations dont l'application a besoin. Ensuite, et particulièrement dans le cas où l'intégration est virtuelle, il est essentiel de profiter des capacités respectives des sources pour optimiser l'accès aux données. Ce second point sera traité en détail dans le chapitre 6. Notons que nous ne traiterons pas des problèmes liés à la mise à jour des données.

Hétérogénéité des formats. Une multitude de formats de représentation est utilisée. Du plus général au plus spécifique, du plus contraignant au plus souple, chacun de ces formats a une raison d'être. Les données utilisées par nos applications sont stockées dans un format bien précis. Ce peut être pour des raisons historiques, mais c'est plus souvent pour les besoins de l'application elle-même (p.ex. dans une base de données pour assurer la persistance, la fiabilité et la concurrence d'accès, en HTML afin de les publier sur la Toile, etc.). Cette multiplicité de formats est l'un des principaux obstacles qu'il nous faut surmonter. Nous y revenons en détails dans la section suivante.

Hétérogénéité sémantique. L'hétérogénéité sémantique [Hei95, KS91] est intrinsèquement la plus difficile à appréhender. Donner une interprétation homogène à un ensemble de sources nécessite la compréhension de chacune d'elles et la résolution de conflits arbitrairement complexes. Dans la suite, nous supposerons que la sémantique d'une source est décrite uniquement par sa structure. Plusieurs arguments permettent de justifier cette hypothèse simplificatrice. Tout d'abord, la structure est souvent la seule information sémantique disponible en pratique. Ensuite, la résolution de l'intégralité des problèmes sémantiques sort largement du cadre de cette thèse. Enfin, nous estimons que le traitement d'autre formes de représentation sémantique (thésauri, ontologies, etc) devrait se révéler largement indépendant du traitement de la structure, tel qu'il est proposé ici.

D'un point de vue purement structurel, l'hétérogénéité sémantique provient essentiellement du fait que la manière de représenter une même information peut changer. En tout état de cause, il n'y a pas une seule bonne façon de représenter une information. Par exemple, le nom d'une personne peut être stocké dans une chaîne de caractères (`"Jules Dupont"` ou `"Dupont, J."`) ou encore dans un n-uplet (`tuple(nom: "Dupont", prénom: "Jules")`). L'âge d'une personne peut être stocké tel quel, ou par l'intermédiaire de sa date de naissance, celle-ci peut être plus on moins précise, etc. Une personne peut enfin être *identifiée* par son nom, par son numéro de sécurité sociale ou bien encore par un identificateur unique, fourni automatiquement par le système.

Ces problèmes, pris dans leur ensemble, concourent à faire de l'intégration une tâche complexe. Avant d'introduire un début de solution dans la section 2.4, revenons un instant sur l'éventail des formats de représentation qu'il faut pouvoir traiter.

2.3 Formats de représentation des données

Le nombre des formats (ou *modèles* dans la terminologie des bases de données) est bien trop important pour en faire une description exhaustive. Nous nous contentons d'en présenter les principales familles, en insistant sur leurs caractéristiques essentielles et sur leurs différences. Comme nous l'avons déjà indiqué, la structure des données, liée à la notion de *typage* du format correspondant, va jouer dans la suite un rôle clé. Pour cette raison, nous choisissons de centrer cette présentation autour des différentes façons de définir et d'utiliser le typage dans ces formats.

Nous commençons par répertorier l'ensemble des *constructions de type* les plus courantes, chaque type étant associé à un ensemble de *valeurs*. Notez que, dans certains cas, une valeur particulière, dite *valeur nulle*, est utilisée pour représenter l'absence d'information ou son incomplétude (le lecteur intéressé pourra consulter [Lib94]).

Les constructions de type élémentaires.

- **Types atomiques :** entier, réel, chaîne de caractères, image (p.ex. gif), etc. Les valeurs associées à ces types portent des informations élémentaires, c'est-à-dire qu'elles ne sont pas interprétées par les langages que nous utiliserons pour les manipuler (voir les travaux sur la notion de généricité dans les langages de requête [CH80, BMT96] pour une étude formelle de cette intuition).

- Les **collections** sont des suites (ensembles, multi-ensembles, listes, tableaux, etc.) de valeurs. Ces valeurs peuvent être du même type (collections homogènes) ou de types différents (collections hétérogènes).

- Les **produits cartésiens** sont des regroupements d'éléments de types différents. On peut y accéder, soit par l'ordre dans lequel ces éléments apparaissent, soit grâce à des noms d'*attributs* (distincts) identifiant chacun d'entre eux.

- **union de type :** décrit un ensemble de types possibles. Dans de nombreux cas, un marqueur est utilisé pour distinguer les différentes alternatives (on parle alors d'*union marquée*).

- **classe/référence :** la notion de référence permet d'identifier une donnée et de partager une information. Ce mécanisme est utilisé de façon intensive dans les modèles hypertextes et orienté-objet. On trouve parfois plusieurs catégories de références : mono ou bi-directionnelles, intra ou inter-documents, etc. Plus particulièrement, dans le modèle orienté-objet, les classes sont utilisées pour le typage des références et indiquent quelles données peuvent être partagées.

Formats avec typage préalable. La grande majorité des formats de représentation s'inscrit dans cette catégorie. Le principe est le suivant : un format offre un certain nombre des constructions de type élémentaires, parmi celles données ci-dessus, ainsi que la possibilité de les combiner afin de définir un *schéma*. Une fois ce schéma fixé, il est possible de l'*instancier*, c'est-à-dire de créer un ensemble de données correspondant à cette structure. Dans ce cadre, le schéma prime toujours sur la donnée : une donnée ne peut être définie que par rapport à ce schéma et ne peut, en aucun cas, le contredire (ou en d'autres termes être mal typée).

Pour illustrer cette idée, voici quelques uns des principaux formats qui optent pour cette approche :

- **Les bases de données**. Dans les systèmes de gestion de bases de données, qu'elles soient relationnelles [Cod70, Ull88] ou orienté-objet [BDK92, Cat97], il faut toujours définir un schéma au préalable. Une base de données est alors une instance de ce schéma. Dans le cas du modèle relationnel, le schéma contient des relations, chacune ayant pour type un ensemble de n-uplets dont les attributs ont des types atomiques. Dans le cas du modèle orienté-objet, un schéma est constitué de classes, le type de chacune d'elles pouvant être une combinaison arbitrairement complexe de types atomiques, de références à d'autres classes, de collections, ou de n-uplets.

- **Les documents structurés**. Le standard SGML [ISO86] (pour *Standard Generalized Markup Language*) est un format très utilisé dans le domaine de l'édition électronique, en particulier pour la gestion de documentations techniques ou les bibliothèques numériques. Bien qu'il ne propose que des chaînes de caractères comme types atomiques, il fournit la quasi intégralité des autres constructions de types que nous avons énumérées (à l'exception des ensembles non ordonnés). Comme pour les bases de données, la structure des documents SGML doit être décrite de façon préalable à l'aide d'une grammaire, appelée *Document Type Definition* ou DTD. Une fois cette grammaire définie, il est possible de créer et de manipuler des documents s'y conformant. HTML (pour *HyperText Markup Language*), dont la DTD peut être trouvée dans [RLJ98], est la plus

```
<! DOCTYPE article1 [
<! ELEMENT article   - - (numéro, titre, auteurs, conf, résumé) >
<! ELEMENT numéro    - - (#PCDATA) >
<! ELEMENT titre     - - (#PCDATA) >
<! ELEMENT auteurs   - - (auteur)+ >
<! ELEMENT auteur    - - (#PCDATA) >
<! ELEMENT conf      - - (#PCDATA) >
<! ELEMENT résumé    - - (#PCDATA) > ] >
```

FIG. 2.3: *Première DTD pour les articles*

illustre famille de documents SGML. La figure 2.3 propose un exemple simple de DTD et va nous permettre d'illustrer les notions de schéma et d'instance introduits jusqu'ici. Cette DTD décrit un ensemble de documents de type article et indique que tous les documents doivent comporter un numéro d'identification, un titre, la liste des auteurs (le symbole '+' indiquant que le nombre d'auteurs est non nul et illimité) le nom de la conférence et un résumé. Seuls les documents se conformant à cette description seront considérés comme valides et pourront être manipulés. La figure 2.4 fournit un exemple de document valide. Chaque élément déclaré dans la DTD est représenté dans le document par un marqueur de début (p.ex. <article>), puis le contenu de cet élément, suivi d'un marqueur de fin (p.ex. </article>).

- **Le format STEP/EXPRESS** est un standard [ISO94a, Owe93, ABCD$^+$94] utilisé dans l'industrie pour l'échange de données techniques. Ce modèle est similaire à un modèle orienté-objet avec de l'union de type marquée.

- **Formats textuels**. De nombreux formats utilisent une simple représentation textuelle des données, décrite par une *grammaire* [Gin66, RS97b]. Les données sont alors stockées dans des fichiers sous la forme de chaînes de caractères respectant précisément la grammaire.

Notons que, dans tous les formats que l'on vient d'énumérer, l'utilisateur peut, soit définir son propre schéma adapté à ses besoins, puis l'instancier, soit utiliser un schéma ou une grammaire déjà définis. Il pourra par exemple utiliser HTML ou l'un des nombreux formats de fichiers existants tels que LaTeX ou BibTeX définis par les grammaires correspondantes.

L'approche suivie dans les formats avec typage préalable possède un certain nombre d'avantages. Connaître par avance le type des données permet en particulier d'en faciliter l'interrogation, d'optimiser le stockage des données et l'évaluation des requêtes. De plus, il est possible de procéder à des vérifications sur les programmes qui manipulent ces données. Il est à noter que les systèmes de gestion de bases de données font un usage intensif de ces caractéristiques. Par contre, il en résulte une certaine lourdeur et un manque de flexibilité. La complexité des méthodologies destinées à mettre au point un « bon » schéma [BEP$^+$91] ainsi que les nombreuses difficultés rencontrées dès que celui-ci doit être modifié [BKKK87] en sont les symptômes les plus frappants.

Formats sans typage préalable. Dans un certain nombre d'applications, ce manque de flexibilité constitue un sérieux handicap. Cela explique l'élaboration de nouveaux formats,

```
<article>
<numéro> 60 </numéro>
<titre> From Structured Documents to Novel Query Facilities </titre>
<auteurs>
  <auteur> Vassilis Christophides </auteur>
  <auteur> Serge Abiteboul </auteur>
  <auteur> Sophie Cluet </auteur>
  <auteur> Michel Scholl </auteur>
</auteurs>
</conf>SIGMOD'94</conf>
<résumé> Structured documents (e.g., SGML) ... </résumé>
</article>
```

FIG. 2.4: *Un document de type article*

pour lesquels les contraintes de type sont plus relâchées. C'est, par exemple, le cas d'ACeDB (pour *A Caenorhabditis elegans Database* [TD92]), un modèle utilisé en bioinformatique pour lequel les contraintes sur la structure des données sont réduites, ou encore d'ASN.1 [ASN94, Rin98], un standard dans le monde des télécommunications utilisé lui aussi en bioinformatique.

Plus récemment, de nouveaux modèles ont vu le jour pour faciliter l'intégration de données hétérogènes, ou pour représenter l'information peu structurée disponible sur la Toile.

- Les modèles *semistructurés* [Abi97, Bun97, Clu97] utilisent des graphes annotés pour représenter les données. La notion de schéma est supprimée et tout graphe peut être considéré comme une donnée valide. Il en résulte une grande souplesse mais une complexité accrue. Nous reviendrons plus en détail sur ces modèles dans le prochain chapitre.

- XML [BPSM98] est une version simplifiée de SGML, dont l'objectif est de pallier les lacunes de HTML tout en restant suffisamment simple pour être utilisable sur INTERNET. Une des grandes différences par rapport à SGML est que dans le cas de XML, la DTD du document est seulement optionnelle. Il est donc possible d'écrire un ensemble de documents sans grammaire préalable. Notez que dans le cas où une DTD est fournie, on se retrouve de nouveau dans le cadre plus rigoureux qui était de mise pour SGML. Si l'on reprend notre exemple, l'article de la figure 2.4 est considéré comme un document XML en l'absence de toute DTD [2].

Bien entendu, l'absence de schéma nous prive des avantages que nous avions cités pour l'approche précédente. En particulier, de nombreuses techniques d'optimisation, qui requièrent une connaissance préalable de la structure, deviennent inapplicables. Par ailleurs, nous avions évoqué le rôle sémantique joué par le schéma : en l'absence de celui-ci il peut devenir très difficile pour l'utilisateur de comprendre les données et de les exploiter. Notez que l'absence de schéma au départ n'indique pas nécessairement une absence totale de structure dans les données. Il n'est pas interdit, ni forcément difficile, d'imaginer un schéma plus ou moins complexe qui corresponde à un ensemble fini de données. Parallèlement, remarquons qu'il n'y

2. Pour autant que ce document soit bien formé, ce qui, *grosso modo*, signifie qu'il respecte certaines conventions lexicales et que le « parenthésage » des marqueurs soit correct. Par exemple, un morceau de document tel que "<article><numero>60</article></numero>" est mal formé donc toujours incorrect.

```
<! DOCTYPE article2 [
<! ELEMENT article   - - ((numéro| isbn), titre,
                          auteurs, conf, résumé, section*) >
<! ELEMENT numéro    - - (#PCDATA) >
<! ELEMENT isbn      - - (#PCDATA) >
<! ELEMENT titre     - - (#PCDATA) >
<! ELEMENT auteurs   - - (auteur)+ >
<! ELEMENT auteur    - - (#PCDATA) >
<! ELEMENT conf      - - (#PCDATA) >
<! ELEMENT résumé    - - (#PCDATA) >
<! ELEMENT section   - - (#PCDATA) > ] >
```

FIG. 2.5: *Seconde DTD pour les articles*

a pas unicité d'un tel schéma. Pour s'en convaincre, il suffit de remarquer que l'article de la figure 2.4 est une instance de la DTD de la figure 2.3, mais aussi de la DTD de la figure 2.5 ! Cette seconde grammaire (qui autorise des numéros ISBN à la place du numéro de référence, et permet d'ajouter des sections à la fin de l'article) est même plus générale que la première : tous les documents valides pour la première le sont aussi pour la seconde, la réciproque étant fausse. Notons aussi que ce dernier exemple illustre une idée qui s'avèrera très importante par la suite : si l'on n'a pas une connaissance complète de la structure que vont avoir les données, il peut être intéressant de fournir une information structurelle, certes moins précise, mais dont on est sûr. Cette connaissance peut être utilisée à nouveau à des fins d'optimisation ou pour guider l'interrogation.

Formats mixtes. Le modèle de données de YAT [CDSS98], décrit en détail dans le chapitre 4, est un modèle semistructuré étendu par des primitives de typage. Ce modèle, très souple, permet de s'adapter à une connaissance plus ou moins forte des données que l'on va manipuler, et d'utiliser cette connaissance au mieux. Il est alors possible de définir un schéma préalable (permettant un maximum d'optimisation), et éventuellement de raffiner *a posteriori* ce schéma en fonction des données elles-mêmes. Nous parlerons ainsi de format mixte, dans le sens où il permet de concilier les deux approches précédentes.

Quelques remarques.

- **Modèles avec contraintes.** L'utilisation de modèles de données fondés sur des systèmes de contraintes semble être une alternative intéressante aux formats plus classiques que nous venons de présenter. En particulier, les *Feature Trees* [Smo92, ST94], conçus originellement pour répondre aux besoins d'applications dans les domaines du traitement de la connaissance ou du langage naturel, proposent un modèle très proche du modèle semistructuré. Ils offrent une grande flexibilité tandis que l'existence d'un système de contraintes supplée l'absence de typage au sens plus classique du terme.

- **Données et programmes.** Quelques formats ne rentrent pas tout à fait dans cette classification. En effet, nous avons supposé jusque là que données et programmes peuvent être séparés et que la sémantique des données peut être capturée indépendamment de celle des programmes qui les utilisent. Ce n'est pas toujours le cas, comme par exemple

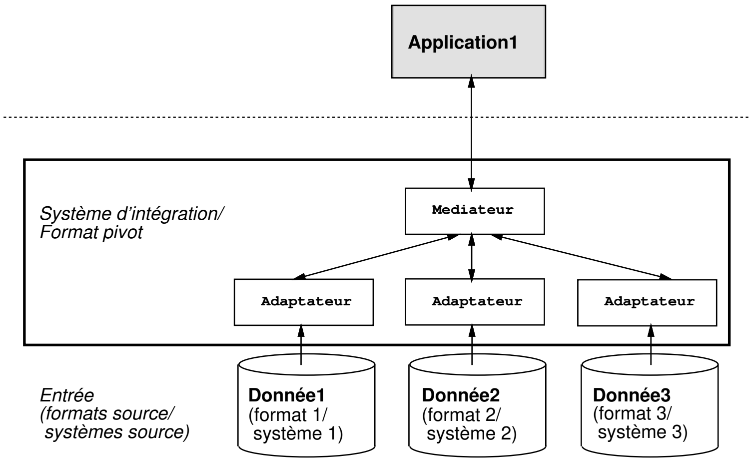


FIG. 2.6: *L'architecture adaptateur/médiateur*

dans les formats TEX, LATEX, postscript™, les documents Word™, ou encore dans les langages de programmation. Dans toute la suite, notre propos ne concernera jamais les programmes eux-mêmes, mais pourra éventuellement être étendu à la partie statique (*i.e.* les données) de ceux-ci, dans la mesure où elle peut être extraite.

- **Données multimédia.** Nous n'avons pas évoqué jusqu'ici les données de nature *multimédia*. Nous supposerons dans la suite que images, sons, vidéos, etc. sont, le cas échéant, manipulés par des outils extérieurs. En d'autre termes, ils seront traités comme des types atomiques.

La variété et la complexité de tous ces formats rendent le travail d'intégration particulièrement ardu. Il faut être capable de manipuler les données qu'elles soient fortement typées ou pas, que la connaissance que l'on en a soit grande ou pas, que les informations évoluent dans le temps ou pas, etc. Ces problèmes reviendront comme un *leit motiv* tout au long de notre travail.

2.4 Une architecture pour l'intégration

Après cet aperçu des difficultés liées à l'intégration de données, voici une première esquisse de solution. Dans le problème qui nous préoccupe, un point est essentiel et orientera toute la suite : pour traiter de données hétérogènes, il est indispensable d'avoir un langage commun pour « parler » de ces données et les manipuler. Ce besoin d'un *format pivot*, déjà évoqué dans l'introduction, conduit naturellement à choisir une architecture, illustrée sur la figure 2.6, similaire à celle proposée par [Wie92]. Les adaptateurs permettent de voir chaque source comme si elle utilisait le format pivot, tandis que le médiateur offre un accès intégré à l'ensemble des informations fournies par les adaptateurs. Notons que, dans la plupart des systèmes d'intégration, l'application finale accède aux données directement dans le format pivot, ce qui ne convient pas toujours (p.ex. lorsque l'on souhaite effectuer une conversion).

Bien que cette architecture soit commune à la totalité des systèmes d'intégration, ceux-ci se distinguent par les trois caractéristiques suivantes : (i) le choix du format pivot, (ii) le degré de généricité des adaptateurs et médiateurs, et (iii) les fonctionnalités respectives offertes par ces derniers.

Choix du format pivot. Le format pivot est utilisé pour la représentation des données provenant de sources hétérogènes. Dans ce contexte, deux critères sont essentiels : la capacité à représenter directement l'ensemble des formats que nous avons énumérés dans la section 2.3 et la flexibilité, afin de s'adapter rapidement aux changements survenant inévitablement dans un contexte hétérogène et distribué.

- **Modèles structurés.** Celui-ci doit naturellement offrir le plus de constructions de types possibles afin de pouvoir représenter directement le plus de formats possibles. Ainsi, le modèle orienté-objet est souvent considéré comme un meilleur candidat que le modèle relationnel.

  Malgré cela, des difficultés persistantes apparaissent lorsque le format à représenter s'éloigne un peu trop du format pivot. Des différences sémantiques finissent toujours par surgir, rendant délicat le choix d'une représentation adaptée (voir par exemple dans [Chr96] les problèmes posés lors de la représentation de documents SGML dans un modèle orienté-objet).

  Plus gênant encore, il est nécessaire de fournir un schéma d'intégration dès le départ, ce qui entraîne deux conséquences néfastes. Primo, cela oblige les adaptateurs à exporter leurs informations par rapport à ce schéma. Ils nécessitent alors un travail de développement important, et ceci à chaque nouvelle intégration. Secundo, les adaptateurs doivent être modifiés chaque fois que les sources où l'application doivent évoluer, ce qui réduit la flexibilité du système.

- **Modèles semistructurés.** Ils offrent beaucoup plus de souplesse dans la mesure où une donnée, quelle qu'elle soit, est toujours valide. De plus, il est très facile de représenter une donnée quelconque sous forme d'arbre ou de graphe, tout en conservant sa structure originelle. En contrepartie, l'absence de typage rend difficile toute vérification ou optimisation.

- **Modèles mixtes.** Pour pallier les inconvénients respectifs des deux approches précédentes, nous opterons dans la suite pour un modèle de données mixte. Nous introduisons le modèle en détail dans le chapitre 4 et montrons que cette solution permet une représentation naturelle et précise des informations contenues dans des sources hétérogènes et offre la flexibilité dont nous avons besoin. Dans le chapitre 6, nous illustrons les opportunités d'optimisation qu'il confère.

Il faut noter que le choix d'un format pivot n'est pas anodin car il détermine en grande partie la complexité des adaptateurs. La capacité du format pivot à représenter un format source a un impact direct sur le travail nécessaire pour la réalisation de l'adaptateur correspondant.

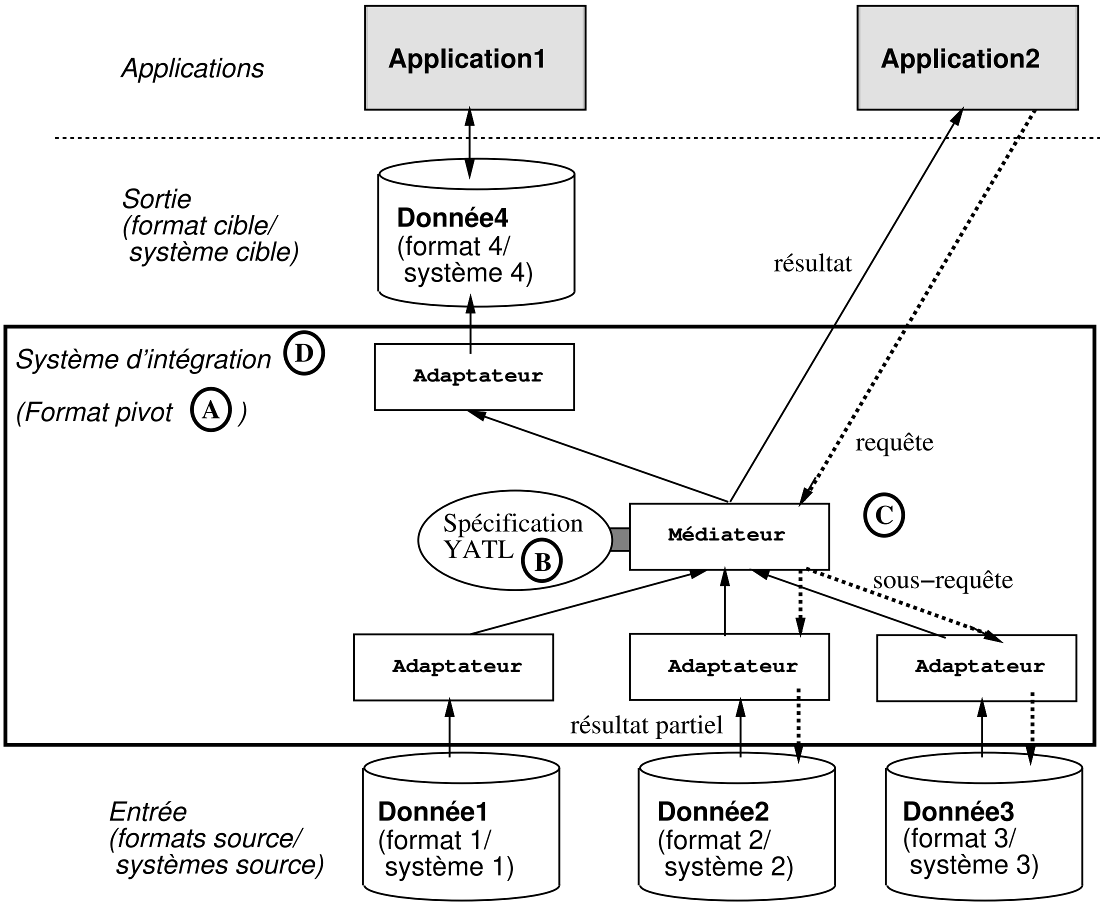
Des composants réutilisables. Le développement du module d'intégration se réduit au développement des adaptateurs et médiateurs. Il est donc essentiel de comprendre dans quelles conditions ces derniers doivent être réalisés. Une caractéristique importante est le degré de généricité de ces composants : plus ils sont génériques, et moins souvent il sera nécessaire de les réécrire pour s'adapter aux évolutions des sources auxquelles ils sont connectés. Aux adaptateurs spécifiques à une source bien précise (p.ex. une base de données avec tel schéma), on préférera donc des adaptateurs spécifiques à une « technologie » (p.ex. un adaptateur pour bases de données objet, un adaptateur pour documents SGML etc.). Cette solution permet d'envisager un environnement où les adaptateurs pour les formats et systèmes sources les plus courants sont écrits une fois pour toute et fournis avec le système.

Nous avons vu que les systèmes semistructurés autorisent de tels adaptateurs. En contrepartie, les médiateurs ne peuvent dans ces systèmes être complètement génériques dans la mesure où ils réalisent la partie sémantique de l'intégration (à l'inverse des systèmes structurés où l'intégration sémantique est codée dans les adaptateurs). Les systèmes semistructurés proposent donc des approches déclaratives pour la description de médiateurs. L'idée est d'offrir un langage, suffisamment simple, puissant et de préférence déclaratif pour décrire l'intégration des données. Il faut remarquer que cela est possible car le médiateur effectue maintenant l'intégration dans un contexte homogène. Dans le chapitre 5, nous présentons YAT$_L$, un langage déclaratif pour la spécification de médiateurs dans les systèmes d'intégration semistructurés.

Fonctionnalités lors de l'évaluation. Lors de leur utilisation par les applications, adaptateurs et médiateurs offrent essentiellement deux types de service : d'un côté ils procèdent au traitement des requêtes soumises au système d'intégration, de l'autre ils convertissent les données issues des sources. Plus précisément, un adaptateur convertit une requête sur le format pivot en une requête compréhensible par la source qu'il encapsule, puis convertit le résultat du format source vers le format pivot. Un médiateur décompose une requête sur la vue intégrée des données en sous-requêtes à envoyer aux adaptateurs, puis intègre les différents résultats.

Grâce à l'utilisation des modèles algébriques hérités des bases de données, les systèmes d'intégration structurés offrent des fonctionnalités avancées pour la décomposition efficace des requêtes. Les systèmes semistructurés souffrent de l'absence de typage de leur modèle pivot. Dans le chapitre 6, nous élaborons, dans le contexte semistructuré, des techniques algébriques qui autorisent un traitement efficace des requêtes.

La figure 2.7 récapitule l'ensemble des composants et fonctionnalités du système YAT et indique pour chacun d'eux le chapitre dans lequel il est traité.



A Le modèle YAT: un format pivot semistructuré avec typage (chapitre 3)

B YATL: un langage d'intégration déclaratif (chapitre 4)

C Évaluation des requêtes et optimisation (chapitre 5)

D Architecture et implantation (chapitre 6)

FIG. 2.7: *Architecture du système YAT et panorama de la thèse*

Chapitre 3

État de l'art

...
Gardez-vous, leur dit-il, de vendre l'héritage
Que nous ont laissé nos parents :
Un trésor est caché dedans.
Je ne sais pas l'endroit; mais un peu de courage
Vous le fera trouver : vous en viendrez à bout.
Remuez votre champ dès qu'on aura fait l'oût :
Creusez, fouillez, bêchez; ne laissez nulle place
Où la main ne passe et repasse.
...

– Jean de LA FONTAINE
(in *Le Laboureur et ses Enfants*)

DANS LE CHAPITRE PRÉCÉDENT, nous avons présenté le contexte et la problématique de l'intégration de données. De plus, nous avons proposé une approche qui permet d'isoler les problèmes mis en jeu et donc de les aborder de façon relativement autonomes. Nous nous intéressons donc maintenant aux différents travaux qui traitent de ces problèmes.

L'organisation de cet état de l'art est la suivante. La section 3.1 fait le bilan des modèles pivots qui ont été utilisés jusqu'ici et introduit le modèle semistructuré. La section 3.2 s'intéresse aux langages permettant de décrire l'intégration proprement dite, tandis que la section 3.3 fait l'étude des travaux concernant l'évaluation des requêtes dans des environnements hétérogènes et distribués. Pour clore cet état de l'art, la section 3.4 tente d'offrir un panorama aussi complet que possible des systèmes conçus pour la réutilisation des données.

3.1 Modèles pour la représentation de données hétérogènes

Comme nous l'avons déjà souligné, le choix du format pivot est crucial. Sans revenir sur les caractéristiques générales des formats possibles, nous présentons les différents travaux utilisant une représentation pivot en nous concentrant sur leurs particularités. En nous inspirant de la classification introduite dans le chapitre précédent, la section 3.1.1 s'intéresse aux modèles fortement structurés et la section 3.1.2 introduit le modèle semistructuré. Dans la section 3.1.3, la présentation des systèmes de types pour ce modèle sera l'occasion de s'orienter vers ce que nous avons appelé les modèles mixtes.

3.1.1 Modèles structurés

Lorsque l'intégration porte sur des sources de données de même nature, c'est bien entendu le modèle commun qui est utilisé comme pivot. C'est ce qui est fait, par exemple, dans les travaux sur les *bases de données réparties* [OV91], les *bases de données fédérées* [Bre90, LMR90, SL90, BHP92] ou les *entrepôts de données* [Wid95, GLWZ98] qui traitent de l'intégration de bases de données relationnelles [Cod70, Ull88]. Le modèle orienté-objet [BDK92, Cat97] a servi de support pour l'intégration [CHS+95], la conversion [Kos96], le chargement en masse [ACD98, Ame99] ou les vues [AB91, Sou95, DST95] dans un contexte impliquant indifféremment des bases de données relationnelles ou à objets. Enfin, SGML [ISO86] a lui aussi été utilisé pour l'intégration de sources documentaires [VDH97, PV98b].

Comme nous l'avons déjà dit, l'utilisation d'un modèle structuré pour l'intégration montre des limites lorsque les sources s'éloignent du format pivot. La structure précise des données étant le support de leur sémantique, il est indispensable de pouvoir représenter chaque structure du format source en une structure du format cible ayant un sémantique similaire. Lorsque cela n'est pas possible, le choix d'une structure alternative (p.ex. l'utilisation de n-uplets ou d'héritage pour la représentation d'union de types [Chr96]) entraîne une perte de sémantique d'autant plus dommageable qu'elle est cautionnée par un modèle fort. Inversement, bien que l'utilisation d'un modèle semistructuré entraîne plus souvent une perte de sémantique dans la mesure où les constructions de type qu'ils proposent sont très pauvres, le choix d'un modèle d'arbres rend cette perte minime et plus facilement compréhensible car la représentation reste naturelle. Enfin, l'approche structurée impose un schéma très précis qui, nous l'avons vu, induit un coût de développement des adaptateurs prohibitif.

3.1.2 Modèles semistructurés

Les modèles dits semistructurés (voir [Abi97, Bun97, Clu97, Suc98] pour une introduction plus détaillée), ont été introduits pour répondre à deux types de besoins applicatifs : (i) fournir un modèle permettant de représenter naturellement des informations très hétérogènes dans le cadre de l'intégration de données [PGW95], et (ii) pouvoir représenter l'information peu structurée et évoluant rapidement que l'on trouve dans les bases de données du génome [BDS95] ou sur la Toile [FFLS97].

L'idée de ces modèles est d'utiliser un graphe orienté et annoté pour représenter les données. Les annotations et l'organisation structurelle du graphe sont alors le support de l'information. Comme nous l'avons déjà évoqué, il n'y a pas de schéma décrivant la structure au préalable et n'importe quel graphe sera considéré comme valide. En contrepartie, les annotations d'un graphe semistructuré peuvent être – et seront souvent – utilisés indifféremment pour représenter des informations habituellement issues d'un schéma ou de ses instances. Notons au passage que cette particularité a pour conséquence intéressante de faciliter la résolution de certains types d'hétérogénéité sémantique (ce sujet sera traité dans la section 5.1.7).

La figure 3.1 donne un exemple de graphe semistructuré dans le modèle OEM (pour Object Exchange Model [PGW95]). Il s'agit d'un graphe dont les nœuds sont identifiés (**&i** identifie le **i**e nœud), dont les arêtes portent des *labels*, et les feuilles des valeurs atomiques. Le *nom* Publications est le point d'entrée de ce graphe, il joue un rôle similaire à celui d'une racine de persistance dans les bases de données. L'absence totale de contraintes sur ce graphe permet, comme on peut le voir, de représenter un ensemble de publications n'ayant que peu de structure commune. Par exemple, les auteurs sont représentés, soit par leur seul nom dans une

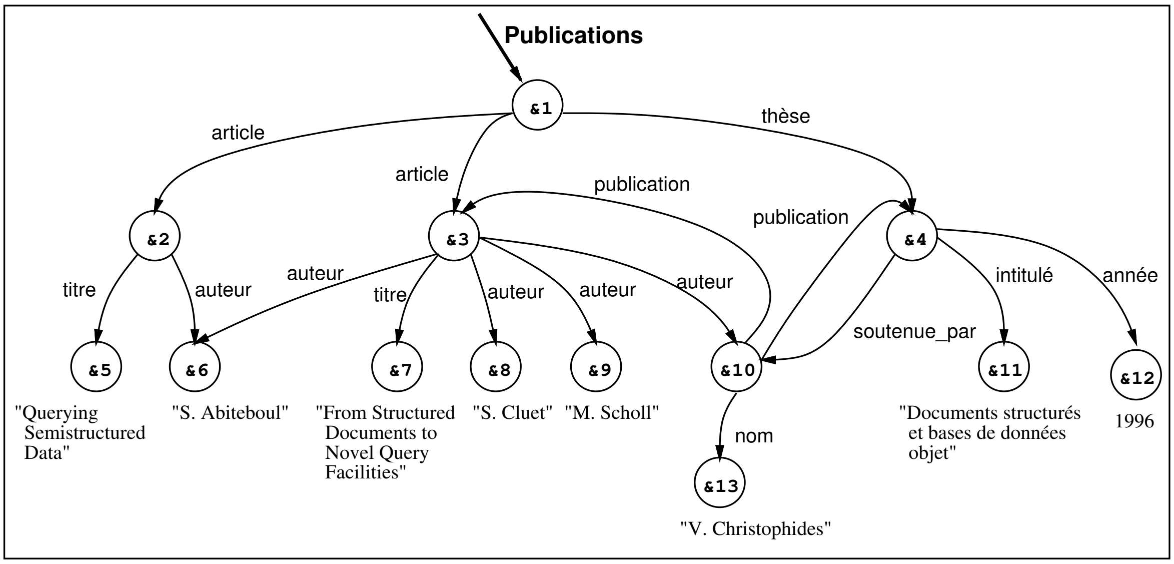


FIG. 3.1: *Un graphe semistructuré OEM*

chaîne de caractères (nœuds **&6**, **&8** et **&9**), soit par leur nom et leurs publications (nœud **&10**). Les publications contiennent le titre et les auteurs (nœuds **&2** et **&3**), ou le titre, les auteurs et l'année de publication (nœud **&4**). Les labels utilisés ne sont pas nécessairement homogènes (cf les labels Titre ou Intitulé), etc. Remarquez que l'utilisation des identificateurs permet le partage d'informations ("S. Abiteboul" est l'auteur des deux articles **&2** et **&3**) et que des cycles peuvent apparaître (entre **&4** et **&10** par exemple).

Cet exemple illustre la capacité à représenter des ensembles d'objets hétérogènes et démontre que le modèle semistructuré répond à nos deux types de besoins : l'hétérogénéité peut provenir du fait que les données sont issues de plusieurs sources (p.ex. les articles **&2** et **&3** viennent d'une base de données à objets, et la thèse **&4** de documents XML) ou du fait que l'information est extraite de pages HTML.

Tous les modèles semistructurés utilisent des graphes annotés pour représenter les données. Quelques différences mineures existent néanmoins entre les modèles proposés, en particulier sur les points suivants : (i) où les annotations sont-elles situées dans le graphe (arêtes et/ou nœuds), (ii) les fils d'un nœud sont-ils ordonnés ou pas, et (iii) comment le partage d'information est-il représenté ? Dans le modèle OEM, les annotations sont, soit des labels situés sur les arêtes, soit des valeurs atomiques situées sur les feuilles du graphe. Les nœuds fils ne sont pas ordonnés, ce qui signifie que dans l'exemple de la figure 3.1, l'ordre des auteurs n'est pas conservé.

Dans [BDS95, BDHS96], il s'agit d'un arbre infini plutôt que d'un graphe, dont seules les arêtes sont annotées (par des labels sur les arêtes internes du graphe ou des valeurs atomiques sur les arêtes terminales). Les nœuds fils sont représentés par des ensembles, c'est-à-dire sans ordre et sans duplicat.

Dans de nombreux formats, notamment les formats documentaires tels que SGML ou HTML, l'ordre conserve une part importante de la sémantique des données. Lors de l'utilisation du modèle semistructuré en tant que pivot, il est important de pouvoir conserver et manipuler cette information. C'est pourquoi [ACM97] propose un modèle d'arbres dont les fils sont ordonnés. Notons cependant que la possibilité d'exploiter cet ordre est également lié

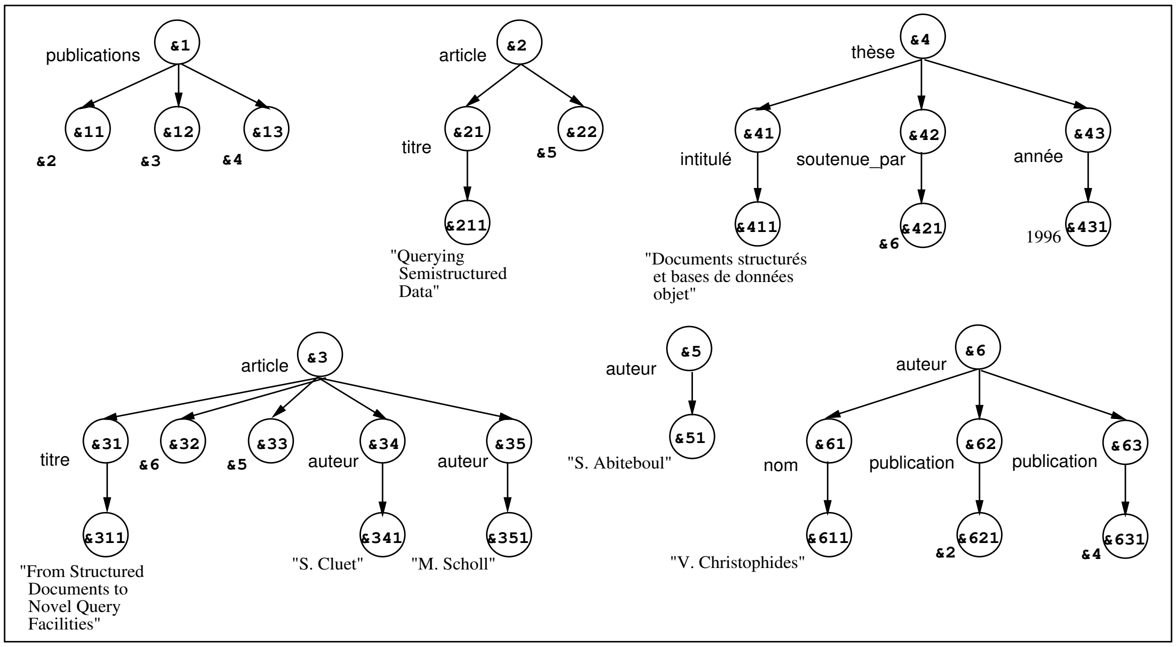


FIG. 3.2: *Un graphe avec labels sur les nœuds et références explicites*

aux opérations susceptibles d'être effectuées sur les données. Elle dépend donc du langage de manipulation qui est défini pour ce modèle : nous reviendrons sur ce point dans la section 3.2. La figure 3.2 illustre comment le graphe de notre exemple est représenté dans ce modèle. Remarquez qu'il s'agit ici d'arbres dont les feuilles peuvent porter des références et non plus de graphes. Par exemple, le nœud **&11** porte une référence à l'article **&2** (ce qui correspond à l'arête [**&1**-**&2**] dans le graphe OEM de la figure 3.1) et l'utilisation de la référence **&6**, commune aux nœuds **&421** et **&32**, modélise le partage d'information. Dans ce modèle, l'ordre des fils est signifiant, ce qui dans l'exemple, permet de conserver l'ordre originel des auteurs de l'article **&3**.

[AM98] s'appuie sur un modèle intermédiaire, où les arbres, utilisés pour représenter des informations de nature hypertexte, ne sont pas annotés par de simple labels mais par des n-uplets. Ce modèle reste cependant très proche d'un modèle semistructuré en raison de l'absence de schéma préalable.

Finalement, XML [BPSM98] est un modèle semistructuré avec de l'ordre et la possibilité d'annoter les nœuds par des n-uplets (pour représenter les attributs d'un élément) et les feuilles par des valeurs atomiques (uniquement des chaînes de caractères).

3.1.3 Typage des données semistructurées

Comme nous l'avons déjà souligné à plusieurs reprises, la flexibilité qu'offrent les modèles semistructurés entraîne une complexité accrue lors de la manipulation des données. Le typage de tels modèles apparaît alors comme un moyen naturel pour résoudre ce problème. Les systèmes de types proposés jusque là diffèrent sur des aspects essentiels tels que la nature des propriétés décrites, ou encore la façon d'obtenir et d'utiliser les informations de typage. Un point commun à toutes ces propositions reste la volonté que le typage ne remette pas en cause la flexibilité du modèle semistructuré.

Partant d'une intuition simple de la notion de type, nous étudions les besoins spécifiques en matière de typage dans un contexte semistructuré. Nous exposons ensuite les propriétés des graphes intéressantes pour le typage et situons dans ce cadre les principaux systèmes de types proposés.

Typer : pourquoi faire ? Intuitivement, un système de types pour un langage donné, est un autre langage, plus « simple » que celui sur lequel il porte, permettant de faciliter le calcul d'une ou plusieurs propriétés. L'exemple le plus célèbre est celui du typage dans les langages de programmation, utilisé pour vérifier des propriétés de correction. Dans le cas général, vérifier la correction d'un programme est un problème indécidable. Le typage est utilisé pour assurer l'absence d'erreurs à l'exécution, propriété importante s'il en est ! Dans le cas des langages de programmation, le système de type est alors tel que : la vérification du type d'un programme donné est, d'une part, décidable et, d'autre part, une condition suffisante de correction.

Dans le contexte des bases de données, le typage est habituellement utilisé pour les raisons suivantes :

- Le type est le support de la sémantique des informations. Il permet d'offrir à l'utilisateur une vision synthétique des informations qu'il peut manipuler et sert de guide à l'interrogation. Remarquons ici que la façon dont le type est obtenu et le fait qu'il soit plus ou moins précis ne sont pas des caractéristiques essentielles. Par contre, la qualité de l'information fournie ainsi que l'adéquation aux données sont des critères importants.

- Il permet d'optimiser le stockage. Il est indispensable de connaître d'avance le type des données pour choisir la représentation physique la plus efficace pour leur stockage. De plus, la création et la maintenance de structures d'accès spécifiques (c.-à-d. d'index : arbres B, tables de hachage, etc) ne peuvent se faire sans connaissance préalable du schéma.

- Il permet d'optimiser l'interrogation. Les nombreuses techniques d'optimisation de requêtes élaborées dans le domaine des bases de données font, elles-aussi, un usage intensif des informations de type (en conjonction avec la connaissance de la représentation physique des données) afin de limiter le nombre d'entrées-sorties.

Deux préoccupations *a priori* contradictoires concourent à la conception d'un système de types pour données semistructurées. D'une part, il s'agit d'imposer certaines contraintes de structures pour permettre une description sémantique plus précise ou l'utilisation de techniques d'optimisation issues des bases de données. D'autre part, il s'agit de conserver la flexibilité qui fait tout l'intérêt du modèle. Notons enfin que dans le contexte de l'intégration, il est important de pouvoir exploiter les informations de types fournies par les sources de données qui sont traitées.

Qu'est-ce qu'un type pour données semistructurées ? En général, un système de types pour le modèle semistructuré est composé d'un langage permettant de décrire des propriétés sur les graphes et d'un ensemble de règles permettant de mettre en relation une donnée avec son type (*i.e.* une expression du langage de type). Les questions qui se posent alors sont les suivantes : quel langage de type choisir ? quelles propriétés permet-il d'exprimer ? et comment ces types vont-ils être calculés (vont-ils être fournis par l'utilisateurs, inférés, calculés à partir des données ou dans un contexte d'intégration, issus des schémas présents sur les sources) ?

Pour répondre à la première question, notons que l'utilisation d'un graphe, pour représenter non seulement les données, mais également les types, est une particularité commune à de nombreuses propositions [NUWC97, BDFS97, CDSS98]. L'intérêt de cette approche est qu'elle permet de décrire un schéma et ses instances dans un même formalisme, la relation de typage étant alors une fonction très simple entre les nœuds de deux graphes. Nous verrons dans le chapitre 4 que cette propriété poussée à l'extrême permet d'offrir un système de types à la fois élégant et flexible.

Quelles propriétés choisir ? Il s'agit maintenant de comprendre quelles propriétés sur les graphes annotés nous souhaitons pouvoir exprimer dans le système de types. Les choix qui seront faits détermineront à la fois la complexité et la puissance du typage. De fait, la plupart des travaux ont cherché à exprimer des propriétés simples sur les graphes.

- **Propriétés des annotations.** Quel est le type de l'annotation d'un nœud ou d'une arête ? Il pourra s'agir directement d'une valeur ou d'un type atomique [NUWC97, GW97, MZ98b], parfois de formules logiques portant sur les valeurs [CDSS98, BM99, BDFS97].

- **Propriétés des branches associées à un nœud.** Il s'agira surtout de décrire les fils d'un nœud, plus rarement de décrire les arêtes qui arrivent à un nœud [NAM98]. Les propriétés décrites porteront sur l'arité (c'est à dire le nombre d'arêtes autorisées et notamment s'il s'agit d'un fils unique ou d'une collection), si les fils sont ordonnés ou pas, et bien entendu quels sont les types de chaque fils. Dans certains cas, il sera nécessaire de décrire précisément tous les fils ou bien on pourra se contenter d'une description partielle (ce qui est similaire, soit dit en passant, à rajouter une relation de sous-typage [Pot98]).

- **Propriétés liées au partage d'information.** Il s'agira enfin d'indiquer quels sont les nœuds du graphe que l'on peut référencer, en d'autres termes, de distinguer quels sont les points de partage autorisés dans le graphe.

Les systèmes de types existants. Observons maintenant comment les différentes propositions se situent dans le cadre que nous venons de mettre en place.

Les *Data Guides* [NUWC97, GW97] offrent un typage très simple dans lequel le type permet de décrire les annotations (par un label ou un type atomique) et l'ensemble des fils possibles à partir d'un nœud. La figure 3.3 donne le type correspondant au graphe OEM de la figure 3.1 et peut se construire en remplaçant les valeurs atomiques par leur type puis en éliminant tous les chemins redondants. Intuitivement, un *Data Guide* est donc un graphe décrivant l'ensemble des chemins possibles (et seulement ceux-ci) à partir de la racine et peut être utilisé pour faciliter l'écriture des requêtes par l'utilisateur [GW97]. Notons que le *Data Guide* est calculé à partir des données elles-mêmes. Cela implique que le « schéma » résultant est toujours correct et que le typage n'interfère en aucune façon avec la flexibilité du modèle semistructuré : quand les données changent, le schéma change avec elles. En contrepartie, le coût nécessaire à la création et à la maintenance d'un *Data Guide* peut se révéler important. De plus, l'utilisation à des fins d'optimisation s'avère difficile d'abord en raison de la pauvreté

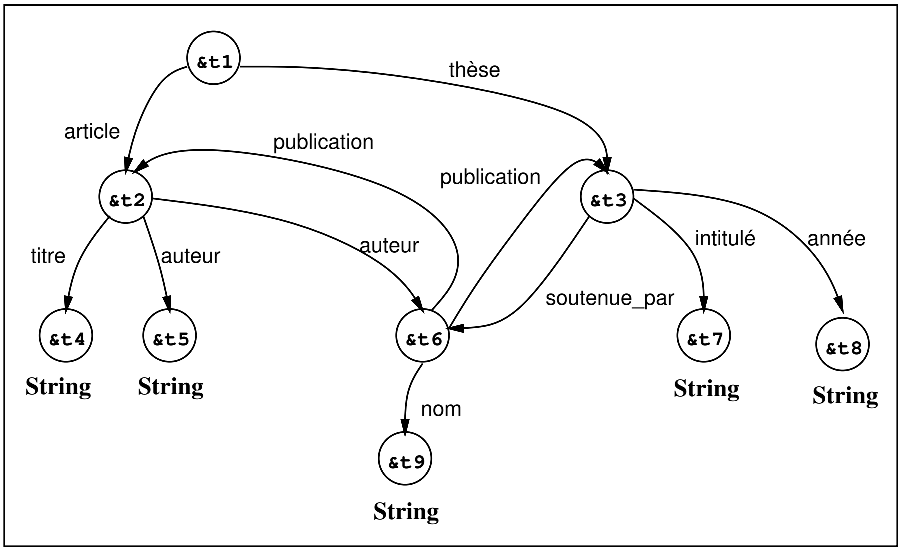


FIG. 3.3: *Un* Data Guide

des informations fournies, ensuite car de nombreuses techniques requièrent une connaissance préalable de la structure.

Les *Graph Schemas* [BDFS97, FS98] se différencient des *Data Guides* par l'utilisation de formules logiques pour décrire les annotation. Ici également, l'information structurelle est modélisée par un graphe ce qui permet de plonger les données elles-mêmes dans le système de types. En conséquence, la relation de *bisimulation* utilisée pour définir le typage d'une donnée peut également servir à capturer une forme d'instanciation (ou *subsomption*) existant entre deux types. Cette relation de subsomption possède une interprétation naturelle relativement au degré de généricité d'un type (un type subsume l'autre tout graphe valide pour le premier l'est aussi pour le second). Ainsi, de la même façon que pour les DTD du document SGML de la figure 2.4, il est possible de choisir un *Graph Schema* plus ou moins précis en fonction de la connaissance que l'on a de la structure des données. Ce modèle rend possible l'utilisation de techniques d'optimisation simples. La provenance de l'information de type et son utilisation dans un contexte pratique n'étant pas indiquées, la complexité qu'implique son obtention ainsi que les interférences possibles avec la flexibilité du modèle semistructuré restent donc incertains.

Le système de types élaboré pour YAT [CDSS98, CS97] est conçu au-dessus du modèle semistructuré proposé dans [ACM97]. Ce modèle permet la description des annotations à l'aide d'expressions logiques et la description des fils d'un nœud vus comme une collection (avec ou sans ordre et avec une information sur l'arité). De plus, l'introduction de deux notions de références distinctes rend possible la description du partage de l'information et de types récursifs. Enfin, ce modèle autorise l'utilisation d'union de types. L'union de types joue un rôle essentiel lorsque les informations structurelles disponibles sont réduites ou lors du typage de données provenant de sources documentaires. Une relation d'instanciation généralise la notion de subsomption des *Graph Schemas* et permet en particulier de modéliser la relation entre un type spécifique et un type plus général. Notons que le modèle a été conçu pour une utilisation dans le contexte de l'intégration de données et favorise l'exploitation des structures fournies par les sources. Enfin, indiquons qu'une formalisation partielle de ce modèle peut être trouvé dans [BM99]. Néanmoins, cette formalisation ne permet pas d'exprimer toute la généralité de la relation d'instanciation.

Parallèlement, le modèle utilisé dans [Zoh97, MZ98a, MZ98b] est une version simplifiée du modèle YAT sans prédicat pour les annotations et sans union de type.

L'approche poursuivie dans [NAM97, NAM98] est originale à plus d'un titre. La description des types est réalisée à l'aide d'un programme datalog [Bid91, AHV95] pouvant être évalué sur les données pour obtenir un typage de celles-ci. Ainsi, l'approche mêle implicitement la description des types et leur calcul à partir des données. Il est possible, dans un certain nombre de cas, d'obtenir des types approximatifs en jouant au cours du typage sur la taille du type et le niveau d'erreur désirés. Enfin, ce modèle permet le typage des arêtes qui arrivent à un nœud du graphe, caractéristique peu courante dans les systèmes de types usuels. Pour toutes ces raisons, la façon dont ce modèle se compare à ceux que nous avons décrits précédemment reste encore à éclaircir. Il ne permet cependant pas de décrire l'arité des collections ou de faire du typage précis.

Utilisation de contraintes. La frontière entre typage et contraintes demeure floue pour plusieurs raisons. Ces notions capturent toutes deux une partie de l'information sémantique. Le typage représente une forme de contrainte dans la mesure où il impose une certaine structure aux données.

À l'inverse, certains systèmes remplacent le typage « classique » des langages de programmation et utilisent uniquement un système de contrainte (voir par exemple les *Feature Trees* dans le domaine de la programmation logique par contrainte [Smo95, Har98]). Enfin, certaines familles de contraintes ont été étudiées pour pallier les lacunes des systèmes de type : contraintes d'intégrité dans les bases de données [Ull88], contraintes de chemin sur des hypertextes [AV97], sur des données structurées [BFW97, BFW98a] ou semistructurées [BFW98b]. On peut néanmoins remarquer que le typage est plutôt vérifié au moment de la compilation tandis que les contraintes sont vérifiées dynamiquement pendant l'exécution du programme.

3.2 Langages de transformation

Nous nous intéressons maintenant aux langages susceptibles de décrire des transformations sur les données. Nous portons naturellement un intérêt particulier aux langages de haut niveau, dédiés et/ou déclaratifs, qui seuls nous permettront de procéder rapidement à une intégration. Nous serons attentifs aux caractéristiques suivantes : le pouvoir de restructuration (pouvoir d'expression), la facilité avec laquelle on peut définir une conversion complexe et la possibilité de maintenir la correspondance entre les données avant et après la transformation.

3.2.1 Langages pour modèles structurés

Nous regardons rapidement les principaux langages pour la manipulation de données dans un contexte structuré. Remarquons que l'approche « orientée schéma » adoptée par ces langages les rend peu appropriés à un usage dans un contexte impliquant plusieurs sources, donc plusieurs schémas. D'ordinaire, ils ne sont pas conçus pour la résolution des conflits sémantiques.

Modèles relationnel et objets complexes. Bien entendu, SQL [Mel88, MS93] est le standard dans le monde relationnel. Par ailleurs, les langages de requête relationnels ont fait l'objet d'études intensives, nous renvoyons le lecteur intéressé à l'abondante littérature sur le sujet ([Ull88] et [AHV95] constituent de bons points de départ).

Quelques langages permettent de faire des restructurations dans des modèles dérivés du relationnel, comme le modèle objets complexes (p.ex. [AH88], LDL [BNR+87], ou encore COL [Gru90, AG87, AGVW89]).

Enfin, SchemaSQL [LSS96b] propose un langage pour spécifier des vues au-dessus de sources hétérogènes, mais dans un contexte purement relationnel.

Modèle orienté-objet. L'originalité du modèle orienté objet provient de la possibilité de construire des structures de données arbitrairement imbriquées (contrairement au modèle relationnel qui n'autorise que des relations « plates ») et à la présence d'identificateurs (identifiants d'objets). Cette originalité a un impact direct sur les langages de manipulation définis sur ce modèle.

On peut grossièrement regrouper ces langages en deux catégories : ceux conçus pour l'interrogation et ceux conçus pour réaliser des restructurations complètes (c'est à dire la création d'une nouvelle base de données). Notons que cette classification reste relativement indépendante du pouvoir d'expression du langage proprement dit (même si un langage de restructuration se doit de fournir certaines fonctionnalités, notamment pour la création d'objets).

Les langages de requête objet (OQL [Clu91, Cat97, Clu98], [SZ89], etc) permettent d'effectuer des transformations éventuellement complexes, mais en une seule expression. Abstraction faite des problèmes liés au schéma dans ce contexte structuré, l'utilisation d'un tel langage pour créer une nouvelle base de données nécessiterait l'écriture de requêtes trop grosses pour être évaluées par le système ou même gérées par l'utilisateur.

Afin de simplifier la mise au point de transformations impliquant l'intégralité d'une base de données, les langages de restructuration favorisent la décomposition des programmes en un ensemble de requêtes (ou de *règles*), chacune décrivant seulement une partie de la transformation désirée. Cela entraîne par la même occasion le besoin d'une sémantique pour plusieurs règles, et en particulier la possibilité de mettre en relation le résultat de chacune d'elles.

Parmi ces langages, on peut citer WOL [Kos96, DK96] ou [BDH+95] qui autorisent la description de transformations entre bases de données à objets ayant des schémas différents. Ces langages sont très expressifs, offrant par exemple la possibilité de définir des transformations récursives, mais difficiles à évaluer efficacement. D'un autre côté, les langages de vues [KK95, AB91, Ber92, Run92, SAD94, GBCG97] permettent non seulement de définir des restructurations complètes, mais aussi de maintenir la correspondance avec les données d'origine. Cela facilite tout particulièrement la répercussion des mises à jour pouvant intervenir sur les données de la base source [KK95] ou de la vue [SLT91, ABS97]. Enfin, les travaux sur le chargement en masse des données [ACD98, Ame97, Ame99] s'intéressent aux compromis entre le pouvoir d'expression du langage de restructuration et l'efficacité de son évaluation.

Quelques études théoriques des langages de manipulation pour les modèles avec identifiants d'objets ont été menées [Mai86, KL89, CW89, KLW95]. On peut noter que certains de ces langages montrent des propriétés de complétude [VVAG97]. En particulier, IQL [AK89] et ILOG [HY90] permettent la définition de transformations dans des modèles de données similaires à ceux des bases de données à objets. IQL offre des caractéristiques plus proches d'un langage de requête, tandis que l'utilisation de *fonctions de Skolem* [1] dans ILOG favorise

1. Dans tous ces travaux, la mise en forme de Skolem [CL94] des formules permet de donner une sémantique claire à la création d'identifiants. L'usage dans ILOG est plus explicite ce qui favorise la spécification précise

la propagation efficace des mises à jour [CH95]. Les fonctions de Skolem permettent, en effet, de conserver la correspondance avec les objets de la base source : leur utilisation semble donc être un choix judicieux dans le contexte des vues (voir aussi [LDB97] à ce sujet).

Notons pour finir que [Ham94] s'intéresse à la résolution des problèmes d'hétérogénéité sémantique lors de la fédération de bases de données à objets, mais ne décrit pas comment le peuplement de la vue intégrée ou l'évaluation des requêtes dans un tel contexte peuvent être réalisés.

Documents structurés. Plusieurs langages de requête pour documents structurés ont été proposés. Si HyQ [ISO92] ou SgmlQL [LMR95] (qui s'inspire de SQL), peuvent décrire des requêtes portant sur des documents SGML, ils ne permettent pas la création de nouveaux documents avec leurs propres DTD.

DSSSL [ISO96], dans un contexte de conversion ou le langage utilisé dans le projet RIO [VP97, PV98b], dans un contexte d'intégration, autorisent la création de nouveaux documents mais optent pour une approche plus procédurale qui limite les possibilités d'optimisation et donc leur usage dans un contexte impliquant de gros volumes de données.

Autres modèles structurés. On trouve plusieurs langages pour la manipulation de documents au format STEP/EXPRESS. Ils sont souvent complexes et de nature procédurale (p.ex. EXPRESS-M [Bai95], EXPRESS-V [Har94]). Le langage VML [Amo94] est plus déclaratif et s'apparente aux langages de vues tels qu'ils sont proposés pour le modèle orienté-objet.

3.2.2 Vers un peu plus de souplesse

Lors de la manipulation de données issues d'un format documentaire, la nécessité d'avoir une connaissance exacte du schéma pour formuler une requête est un sérieux handicap. Pour cette raison, certains travaux récents tentent de réduire les contraintes liées au schéma.

Langages avec expressions de chemin généralisées. La notion d'*expression de chemin généralisée* [KKS92] est introduite pour permettre la description de parcours dans les données sans pour autant avoir une connaissance précise de leur structure. POQL [CACS94, Chr96] étend OQL avec de telles expressions de chemin dans le but de faciliter l'interrogation de documents structurés stockés dans une base de données à objets. Par exemple, pour trouver toutes les références bibliographiques contenues dans la troisième section d'un article, on peut utiliser l'expression de chemin "a.sections[3] @P.référence" qui dénote toutes les informations accessibles à partir de l'article a, en passant par un attribut sections de type liste, dont on prend le 3e élément, suivi d'un chemin quelconque dans les données (@P), suivi par un attribut référence. Il n'est alors plus indispensable de connaître précisément la structure des sections. Notons que le typage des requêtes POQL requiert une extension du système de types utilisé dans les bases de données à objets [Chr96]. Les expressions de chemin généralisées capturent une forme simple de récursion, ce qui, hélas, favorise l'écriture de requêtes coûteuses à évaluer car nécessitant le parcours d'une grande partie de la base. La mise au point de techniques d'optimisation dans ce contexte reste un problème ouvert [Chr96, CCM96, TGS96, CCMS98, MS99].

---

des créations d'objets.

Interrogation d'hypertextes. Les formats d'hypertextes, au premier rang desquels on trouve HTML, vont nous servir de transition entre interrogation structurée et semistructurée. Conçu comme une application de SGML, HTML offre, tout d'abord, une structure définie par une DTD très précise (cf. [Rag97] ou [RLJ98]). Malheureusement, les structures décrites (titre, liste, vecteur, etc) et l'utilisation abondante de l'union de type donnent à cette DTD un caractère générique, ce qui réduit dramatiquement la sémantique des documents.

De nombreux langages pour l'interrogation d'hypertextes (p.ex. GraphLog [CM89] ou GRAM [Ama94]) et plus récemment de documents HTML (p.ex. WebSQL [MMM97], WebLog [LSS96a] ou encore W3QL [KS95, KS98]) ont été proposés. Suivant qu'ils s'intéressent plutôt à la structure du documents (c.-à-d. aux marqueurs SGML et à leur organisation), au contenu textuel, ou au graphe d'hyper-liens qui est défini, ces langages se rapprochent d'un langage structuré ou semistructuré. Ils proposeront, par exemple, des fonctionnalités similaires à SQL ou OQL pour l'interrogation de la structure, des expressions de chemin généralisées pour l'interrogation du graphe de liens et l'utilisation d'index plein textes pour l'interrogation de contenu textuel. La frontière est définitivement franchie avec WebOQL [AM98] qui permet la restructuration de sites HTML vus comme des graphes semistructurés.

3.2.3 Langages pour modèles semistructurés.

Plusieurs langages pour la manipulation de données semistructurées ont vu le jour au cours des dernières années. En fonction de la nature des applications pour lesquelles ils ont été conçus (interrogation, intégration, conversion, gestion de sites sur la Toile, etc), ces langages diffèrent sensiblement dans les services qu'ils proposent. En particulier, les fonctionnalités suivantes nous serviront à distinguer les différentes approches : la gestion de la récursion, le traitement des collections, la création d'objets et la façon de gérer le manque d'informations relatives au schéma.

Une approche fonctionnelle pour la manipulation de graphes non structurés est proposée dans [BDS95]. L'idée directrice, fondée sur la notion de *récursion structurelle* [BTBN91], est d'appliquer récursivement une même fonction sur tous les nœuds d'un graphe afin de le transformer. Il s'agit d'un procédé plutôt explicite, mieux adapté à l'implantation qu'à la définition d'un langage de requête utilisateur. C'est pourquoi UnQL [BDHS96] choisit une syntaxe proche de SQL et cache la récursion structurelle à travers l'utilisation d'expressions de chemin généralisées.

Lorel [QRS+95, AQM+97] est un langage de requête pour données semistructurées dont la syntaxe est inspirée d'OQL. Lorel utilise de puissantes expressions de chemin généralisées et des règles pour la conversion automatique entre types afin de compenser les erreurs que pourrait faire l'utilisateur en raison de l'absence de schéma. Il s'agit plutôt d'un langage pour l'interrogation que d'un langage pour décrire des restructurations complexes sur les graphes. Pour cette raison, [AMR+98] étend Lorel avec des primitives de création d'objets et la possibilité de cacher des informations dans le but de définir des vues semistructurées.

MSL [PGU96, PAG96] est un langage utilisé pour la spécification de médiateurs. Un programme MSL est composé d'un ensemble de règles similaires à des requêtes Lorel mais sans expression de chemin généralisée. MSL offre la possibilité d'utiliser des fonctions de Skolem explicitement dans les requêtes, ce qui permet de spécifier les relations entre les données créées par les différentes expressions du programme. Il est possible d'appeler des fonctions externes pour le calcul de nouvelles valeurs, fonctionnalité essentielle pour le traitement de l'hétérogénéité sémantique. Par contre, MSL n'offre pas de primitives pour la récursion.

StruQL [FFK+97, FFK+98] (ou son *alter ego* XML-QL [DFF+98]) est un langage, également à base de règles, conçu dans le contexte de la création de sites INTERNET. Un programme StruQL autorise l'usage d'expressions de chemin généralisées, des fonctions de Skolem explicites mais pas de fonctions externes.

Notons qu'aucun de ces langages n'offre la possibilité de manipuler des collections ordonnées ou de fonctionnalités liées au typage. Les expressions de chemin généralisées jouent un double rôle en permettant d'une part, l'interrogation sans un connaissance précise de la structure, et d'autre part, en capturant une forme de récursion. Enfin, remarquons qu'en l'absence totale de schéma, comprendre la structure des données peut, quand même, demeurer difficile, nécessiter de nombreuses requêtes et des expressions de chemin de plus en plus complexes.

Parallèlement, [ACM97] propose un langage pour définir des transformations de données qui recèle plusieurs originalités. Tout d'abord, il est possible de manipuler différents types de collections (avec ou sans ordre, avec ou sans duplicats). Ensuite, il permet dans certains cas d'extraire automatiquement un programme de transformation à partir d'informations sur les correspondances entre graphes d'entrée et de sortie. Cette dernière possibilité permet d'envisager la propagation de mises à jour à la manière des vues. Par contre, l'utilisation des fonctions de Skolem n'étant pas explicite, cela rend délicat la spécification des références dans le graphe généré (implicitement prise en charge par la sémantique de point fixe adoptée).

YAT$_L$, décrit dans le chapitre 5, étend le langage de transformation présenté dans [ACM97] avec des fonctions de Skolem explicites, des primitives de plus haut niveau pour la manipulation des collections (regroupement et ordonnancement), du typage et des mécanismes de personnalisation.

3.3 Modèles opérationnels et évaluation des requêtes

La plupart des langages de manipulation de données que nous avons cités dans la section précédente sont de nature déclarative c'est à dire qu'ils permettent de décrire une transformation (le « quoi ») sans indiquer précisément la façon de l'évaluer (le « comment »). Il en résulte plus de simplicité lors de l'écriture des transformations. En contrepartie, trouver une bonne politique d'évaluation est à la charge du système.

Les systèmes de gestion de bases de données proposent des techniques d'optimisation favorisant l'évaluation efficace des langages de requêtes déclaratifs dans un contexte impliquant de grandes quantités d'information. Une fois de plus, cette section adopte une organisation subordonnée aux différents formats de représentation. Pour chacun d'eux, nous nous intéressons aux modèles opérationnels qui leurs sont associés et aux techniques d'optimisation supportées. Nous étudions ensuite les techniques spécifiques à un contexte distribué et hétérogène.

3.3.1 Techniques d'optimisation relationnelles

Les systèmes de gestion de bases de données relationnels disposent de techniques d'optimisation qui ont fait leurs preuves [Ull88]. La plupart des systèmes se fondent sur une algèbre constituée d'opérateurs relationnels simples (sélection, projection, jointures, etc.) qui est utilisée pour décrire l'évaluation des requêtes. Les équivalences supportées par ces opérateurs induisent des réécritures algébriques conservant la sémantique mais potentiellement plus simples à évaluer. Le problème est alors de trouver parmi toutes les réécritures possibles celle qui est la plus efficace.

Ceci étant dit, quelques remarques s'imposent. L'optimisation des requêtes nécessite la compréhension simultanée d'aspects logiques (choix des réécritures algébriques), physiques (choix du meilleur algorithme pour évaluer un opérateur de l'algèbre en fonction des structures d'accès aux données) et systèmes (impact de la taille de la base, de la mémoire disponible, etc). En outre, le nombre de réécritures possibles pour une requête donnée est souvent trop élevé pour les parcourir toutes. Tout cela fait de l'optimisation un problème complexe qui a fait l'objet de nombreux travaux [Gra93].

Pour simplifier le choix d'une évaluation raisonnablement efficace, les systèmes utilisent à la fois des stratégies de recherche avec modèles de coût (*i.e.* la génération d'un grand nombre de réécritures, suivi par l'estimation du coût des différentes alternatives) et des heuristiques dans le but de réduire cet espace de recherche. Parmi les heuristiques couramment utilisées, on peut citer : pousser les sélections et les projections (pour éliminer, dès que possible, les données inutiles), l'utilisation des chemins d'accès (pour tirer avantage des index qui existent sur certains chemins), le réordonnancement de jointures multiples (pour réduire la taille mémoire nécessaire à l'algorithme de jointure) ou encore l'exploitation de contraintes d'intégrité.

L'utilisation de ces techniques dans un contexte distribué ne fait que compliquer encore les choses [OV91]. Une des techniques les plus intéressantes consiste à exploiter les capacités respectives des différents sites sur lesquels les données sont distribuées, dans le but de réduire le coût global de l'évaluation ou le temps de réponse (voir par exemple [BC81, KYY82, LMH$^+$85]). Les bases de données fédérées étendent ces techniques dans le cadre de l'intégration virtuelle de sources relationnelles [Bre90, LMR90, SL90, BHP92], tandis que les entrepôts de données s'intéressent à l'optimisation dans un cadre matérialisé [WGL$^+$96, Huy97] ou partiellement matérialisé [HZ96].

Dans un contexte complètement hétérogène, il est souvent plus difficile d'exploiter les capacités des sources à intégrer dans la mesure où celles-ci peuvent ne pas supporter les opérateurs relationnels ou seulement une partie d'entre eux. Le système DISCO [TRV96] fournit, outre des techniques d'optimisation standard, la possibilité de décrire les opérateurs relationnels supportés par une source [KTV97] donnant à l'optimiseur l'opportunité d'en tirer parti, ainsi qu'un modèle de coût extensible [NGT98] permettant d'intégrer dans la stratégie de recherche les paramètres spécifiques aux sources.

Notons enfin que des techniques spécifiques pour les jointures entre bases de données et sources documentaires sont présentées dans [CDY95].

3.3.2 Techniques d'optimisation orienté-objet

Les systèmes de gestion de bases de données à objets et les systèmes d'intégration utilisant un modèle pivot orienté-objet proposent des techniques d'optimisation similaires.

Plusieurs algèbres objet ont été proposées [SZ89, KM94, CM93], permettant l'adaptation des techniques d'optimisation relationnelles à ce nouveau contexte. Pour obtenir cela, ces algèbres proposent des opérateurs permettant d'extraire les informations nécessaires à partir des structures de données objet (p.ex. *Image* dans [SZ89], χ dans [KM94] ou *Map* dans [CM93]). Les informations extraites sont retournées dans des relations, ce qui permet ensuite d'utiliser les opérations algébriques et les techniques d'optimisation relationnelles. En outre, de nouvelles techniques sont élaborées, notamment pour l'évaluation efficace de requêtes imbriquées [CM93], cas très courant dans le contexte des objets.

L'optimisation dans le cadre des vues objet a été peu étudiée jusqu'ici. L'optimisation de programmes de transformation impliquant un ensemble de règles s'évaluant dans une algèbre objet est étudiée dans [ACD98, Ame99]. Ce travail ouvre de nouvelles perspectives pour l'évaluation efficace des langages de vues.

De même, les travaux sur la fédération de bases de données à objets s'attachent à exploiter les techniques d'optimisation distribuées issues du relationnel [YOL97, LPL96]. L'exploitation des capacités de sources hétérogènes dans un contexte objet est étudié dans [RS97a, HKWY97]. Notons que, dans tous les cas, ces techniques demandent le développement d'adaptateurs spécifiques pour les différentes sources, et par conséquent un travail important.

Enfin, certains travaux s'intéressent aux techniques d'optimisation spécifiques adaptées à des bases de données à objets stockant des documents [ACC+97, Chr96, CCM96, CCMS98, ACM98].

3.3.3 Techniques d'optimisation semistructurées

Tout se complique dans un contexte semistructuré. En effet, les techniques que nous venons de voir font un usage intensif du schéma : que ce soit pour la mise en place et la maintenance des structures d'accès ou lors des réécritures algébriques (p.ex. certaines techniques fonctionnent sur les ensembles mais pas sur les listes). Par ailleurs, l'absence de schéma conduit à l'utilisation d'expressions de chemin généralisées alors que c'est justement le schéma [Chr96, CCM96, CCMS98, FPS97, BM99, MS99] qui donne la seule opportunité raisonnable d'optimisation de ces expressions !

C'est pourquoi l'évaluation de requêtes pour données semistructurées a été l'une des motivations principales pour l'élaboration de systèmes de types dans ce contexte [BDFS97, GW97, FS98].

L'évaluation de Lorel repose sur une algèbre et des techniques d'optimisation relationnelles [AQM+97]. Il en va de même pour StruQL [FFK+98]. L'évaluation de UnQL [BDFS97] repose sur la récursion structurelle, un paradigme puissant mais complexe à optimiser. [CM96] propose une approche intermédiaire et s'intéresse aux familles de requêtes qu'il est possible d'évaluer en un seul parcours d'un graphe semistructuré. Notons que [AMR+98] et [FFLS98] s'intéressent aux techniques pour la matérialisation efficace de vues semistructurées.

L'évaluation de requêtes dans un contexte d'intégration est étudiée dans [PAG96], mais les techniques d'optimisation proposées sont d'ordre syntaxique ce qui limite leur portée. La possibilité d'exploiter les capacités des sources est envisagée dans [HGN+97, PGGU95]. Malheureusement, les descriptions des sources sont spécifiques (*i.e.* il n'est pas possible de décrire les capacités complètes de SQL, mais seulement des capacités relatives à un schéma donné) et doivent donc souvent être écrites pour une nouvelle intégration. Ainsi il n'est plus possible d'utiliser des adaptateurs génériques dont on a vu qu'ils constituaient l'un des principaux avantages des systèmes d'intégration semistructurés.

Enfin, [CCS98] propose une algèbre pour données semistructurées qui permet l'utilisation des techniques d'optimisation standard, la description précise des capacités opérationnelles de sources hétérogènes et des techniques pour l'optimisation distribuée dans ce contexte. Tout ceci est décrit dans le chapitre 6.

3.4 Systèmes pour la réutilisation des données

Dans cette dernière section, nous essayons de faire un bilan aussi complet que possible des principaux systèmes (prototypes de recherche ou produits industriels) qui existent à l'heure actuelle et permettent la réutilisation des données. Nous partons des systèmes les plus simples permettant, par exemple, la conversion automatique de documents d'un format à l'autre, pour aboutir aux systèmes d'intégration généralistes (voir la classification proposée au tout début du chapitre 2). Nous ne reviendrons pas sur les techniques sur lesquelles ils reposent que nous avons déjà présentées, mais nous nous intéresserons aux fonctionnalités offertes du point de vue de leur utilisation.

3.4.1 Systèmes pour la conversion

De nombreux outils plus ou moins génériques ont été conçus pour réaliser des conversions. En voici quelques uns, du plus simple au plus évolué.

Convertisseurs simples. On trouve de nombreux outils pour la conversion entre formats de fichiers (p.ex. rtf2LaTeX, LGrind [LGr98]). En particulier, il existe beaucoup de convertisseurs vers HTML, et ceci pour la plupart des formats de documents (par ex. LaTeX [Dra94, Mar98], BibTeX, Word). Ces outils offrent une conversion unique, et très peu de moyen de la modifier pour ses propres besoins. Dans de nombreux cas, l'utilisateur doit donc éditer le résultat à la main, ce qui pose naturellement des problèmes lorsque les deux versions d'un même document doivent être maintenues. Dans le meilleur des cas, l'utilisateur peut redéfinir tout ou partie de la conversion en utilisant un langage de programmation. Dans O_2Web [O2W96], la transformation des objets d'une base de données en pages HTML est effectuée, soit de façon générique, soit codée explicitement par l'utilisateur en C++. Enfin, il est impossible d'intégrer plusieurs documents avec ces outils.

Outils pour le chargement. Des outils pour le chargement de bases de données ont été développés. On trouve des exemples du relationnel vers l'objet [O2D96], pour le chargement vers une base de données à objets, de fichiers textuels [ACM93, ACM95] en général ou de documents SGML [Chr96] en particulier. Morphase [2] [Kos96, DK97], conçu autour du langage WOL, permet de décrire et de réaliser des conversions entre bases de données à objets. Ces systèmes offrent des performances limitées et il faut souvent repartir de zéro lorsqu'un incident survient. Le système RelOO [Ame97, Ame99] permet le chargement de bases de données objet, en insistant sur les problèmes de flexibilité et de performance.

Systèmes génériques. Plus récemment, à la suite des travaux théoriques de [ACM97], des systèmes destinés à spécifier et implanter des modules de conversion ont été développés. Le système W3Trans [3] [Mog96] en est une implantation directe. Le système TranScm [4] [Zoh97, MZ98a, MZ98b] démontre que la spécification d'une conversion peut dans certains cas être automatisée.

3.4.2 Systèmes de vues

Les bases de données relationnelles commerciales offrent presque toutes un mécanisme de vue, SQL étant toujours le langage sous-jacent.

2. http://www.cis.upenn.edu/~db/morphase/
3. http://www.math.tau.ac.il/~milo/trans/trans.html
4. http://www.math.tau.ac.il/~sagit/tranScm/

MultiView[5] [KR96] et O_2Views[6] [SAD94, Sou95] permettent de définir des vues au-dessus de bases de données à objets. O_2Views permet, en outre, de répercuter des mises à jour effectuées sur la vue [ABS97].

3.4.3 Gestion de sites sur la Toile

Les problèmes de gestion de données sur INTERNET ont entraîné l'émergence de nombreux systèmes conçus pour faciliter la gestion de sites. ARANEUS[7] [AMM97, AMM98] utilise des bases de données relationnelles pour le stockage des informations et de nombreux outils pour la description de sites. RIO[8] [VP97, PV98b] permet la définition de documents HTML « virtuels ».

Strudel[9] [FFK+97, FFLS97, FFK+98] et WebOQL [AM98] sont des systèmes de gestion de sites fondés sur les technologies semistructurées.

3.4.4 Systèmes pour l'intégration

Systèmes structurés. On trouve de nombreux systèmes fondés sur les technologies relationnelles. Il peut s'agir de bases de données fédérées (p.ex. Multibase [LR82]) ou de systèmes d'intégration à vocation généraliste (Disco[10] [TRV96]). On trouve également de nombreux systèmes industriels offrant des fonctionnalités similaires [Com].

Dans un contexte orienté objet, la même approche est poursuivie par AURORA [YOL97], Pegasus [ADD+91], IRO-DB [GGF+95], InterBase [MBE95], DIOM [LPL96] ou bien encore Garlic[11] [CHS+95]. Garlic offre les fonctionnalités les plus avancées en matière d'optimisation distribuée, permettant par exemple l'intégration des capacités opérationnelles de sources textuelles.

Une fois de plus, tous ces systèmes imposent le développement d'adaptateurs à chaque nouvelle intégration, ce qui représente un travail important.

Systèmes semistructurés. TSIMMIS[12] [PGW95, HGN+97] est un prototype de système d'intégration semistructuré. Les fonctionnalités du système sont réduites par la pauvreté des techniques d'optimisation (due à l'absence de schéma et à l'impossibilité de décrire les capacités opérationnelles des sources à un niveau générique) et de la représentation sémantique offerte à l'utilisateur (due une fois encore à l'absence de schéma).

Le système YAT est décrit en détail dans le chapitre 7.

Systèmes pour l'intégration sémantique. Plusieurs systèmes se concentrent sur les aspects sémantiques de l'intégration. Parmi ces systèmes, on peut citer notamment Information Manifold [LRO96], OBSERVER [MIKS99], InfoSleuth [BBB+97], Infomaster [GKD97], SIMS [ACHK93] ou encore HERMES [SAB+95].

---

5. `http://www.eecs.umich.edu/~rundenst/group/multiview-project.html`
6. `ftp://ftp.inria.fr/INRIA/Projects/verso/GoodStep/O2Views`
7. `http://poincare.inf.uniroma3.it:8080/Araneus/`
8. `http://star.vic.cmis.csiro.au:8042/RIO/`
9. `http://www.research.att.com/~mff/strudel/`
10. `http://www-rodin.inria.fr/disco/`
11. `http://www.almaden.ibm.com/cs/garlic/`
12. `http://www-db.stanford.edu/tsimmis/`

Chapitre 4

Modèle de données et filtrage

...
Auprès de mon arbre,
Je vivais heureux
J'aurais jamais dû m'éloigner d' mon arbre
Auprès de mon arbre,
Je vivais heureux
J'aurais jamais dû le quitter des yeux.
...
– Georges BRASSENS
(Auprès de mon arbre, in *Chanson pour l'auvergnat*, 1956)

LE MODÈLE PIVOT DOIT FOURNIR LE MOYEN DE REPRÉSENTER l'information hétérogène à manipuler lors de l'intégration. Nous avons fait le choix d'un modèle semistructuré étendu par des primitives de typage, afin de fournir les fonctionnalités suivantes : souplesse et richesse de la représentation d'une part, contrôle et optimisation d'autre part. Ce chapitre a pour but de présenter le modèle de données YAT et de démontrer les possibilités de typage qu'il procure. De plus, nous présentons la notion de filtrage. Bien que le filtrage soit utilisé principalement dans le langage d'intégration décrit dans le chapitre 5, nous l'introduisons ici en raison de ses liens avec le typage.

Nous décrivons le modèle de données dans la section 4.1 puis le filtrage dans la section 4.2. Une présentation plus formelle de ces notions est ensuite proposée dans la section 4.3. Notons que, dans ce chapitre, nous nous attacherons surtout à illustrer les capacités de représentation du modèle, tandis que les opportunités de contrôle et d'optimisation seront exploitées respectivement dans les chapitres 5 et 6.

4.1 Modèle pivot : des données vers les types

Comme nous l'avons déjà vu dans la section 3.1.2, les modèles semistructurés n'utilisent pas de schéma et représentent uniquement des *données concrètes*. L'originalité du modèle YAT est qu'il permet de représenter de façon naturelle des informations à différents niveaux d'abstraction. Pour ce faire, il étend le modèle semistructuré proposé dans [ACM97] par des primitives de type simples. Dans les sections 4.1.1 et 4.1.2, nous présentons les éléments syntaxiques du modèle permettant, respectivement, la représentation des données concrètes, et la représentation d'informations plus abstraites. Puis, nous montrons dans la section 4.1.3 comment ces éléments capturent deux formes importantes de typage. Nous insistons sur les bonnes propriétés du système de types obtenu, en particulier, comment le modèle permet de raisonner sur le degré de généricité.

4.1.1 Représenter les données concrètes

Généralités. En fournissant une représentation uniforme d'un ensemble de données hétérogènes, le modèle pivot sert de support à la manipulation des informations présentes dans les sources. Il doit donc être suffisamment flexible pour que la représentation se fasse simplement et demeure intuitive.

Il faut malheureusement admettre qu'une perte d'information sémantique est inévitable lors du processus consistant à convertir une donnée d'un modèle dans un autre. Dans ce contexte, il est indispensable que le modèle pivot limite cette perte d'information pour que l'utilisateur puisse facilement rendre aux données leur sémantique originelle. Nous resterons attentifs à ce problème dans la suite de cette présentation. Notamment, nous expliquerons comment une utilisation judicieuse du modèle assure que l'information sémantique puisse être aisément retrouvée ultérieurement, en particulier, lors de la description de l'intégration.

Le modèle. Le modèle de données YAT est inspiré de celui proposé dans [ACM97]. Il s'agit d'un modèle constitué d'arbres nommés (ou *motifs*), dont les nœuds sont annotés par des *labels* et dont les fils sont des collections, au choix, ordonnées avec des duplicats (liste), non ordonnées avec des duplicats (multi-ensemble) ou non ordonnées sans duplicat (ensemble). Notons que pour ne pas multiplier les notations, nous n'utiliserons dans la suite que les ensembles et les listes. Les motifs peuvent être rassemblés en *forêts* et contenir des références autorisant ainsi la modélisation de structures cycliques et le partage d'informations. Dans la suite, nous utiliserons plusieurs syntaxes pour les motifs : une syntaxe formelle qui sera introduite dans la section 4.3, la syntaxe implantée dans le système YAT, et une syntaxe graphique qui, plus intuitive, sera utilisée dans de nombreux exemples. Dans cette dernière, nous notons les nœuds avec ordre (listes) par le symbole $\bullet$ et les nœuds sans ordre (ensemble) par le symbole $\circ$.

La figure 4.1 donne l'exemple d'un document XML référençant trois articles de conférences (le deuxième d'entre eux étant celui de la figure 2.4) et illustre leur représentation par une forêt YAT. Les nœuds sont annotés par des valeurs atomiques (provenant ici du document d'origine, p.ex. "Vassilis Christophides", 60) ou par des symboles (correspondant aux noms des marqueurs XML, p.ex. auteurs, titre). Les motifs portent un *nom* (***document_xml***, ***article*1**, etc) qui sert d'identifiant et peut être utilisé en référence (p.ex. **&*article*1** dans le document principal est une référence à l'arbre correspondant à l'article n° 1). Les références servent à partager une information : ainsi, l'***auteur*1** est commun aux trois articles.

Contrairement à [ACM97], les fils d'un nœud peuvent être ordonnés (listes) ou non ordonnés (multi-ensembles). Autre différence, seuls les nœuds explicitement nommés peuvent être référencés et partagés. Ces quelques écarts par rapport au modèle proposé dans [ACM97] sont introduits dans le but de capturer plus fidèlement les constructions provenant de formats variés. Par exemple, le modèle orienté-objet utilise des identifiants et distingue les données pouvant être référencées (objets) des autres (valeurs). Si l'on revient à notre exemple, remarquons que tous les nœuds sont ordonnés ($\bullet$) la structure des motifs reflète précisément celle du document XML : les sous-éléments sont les fils de l'élément qui les contient. Bien entendu,

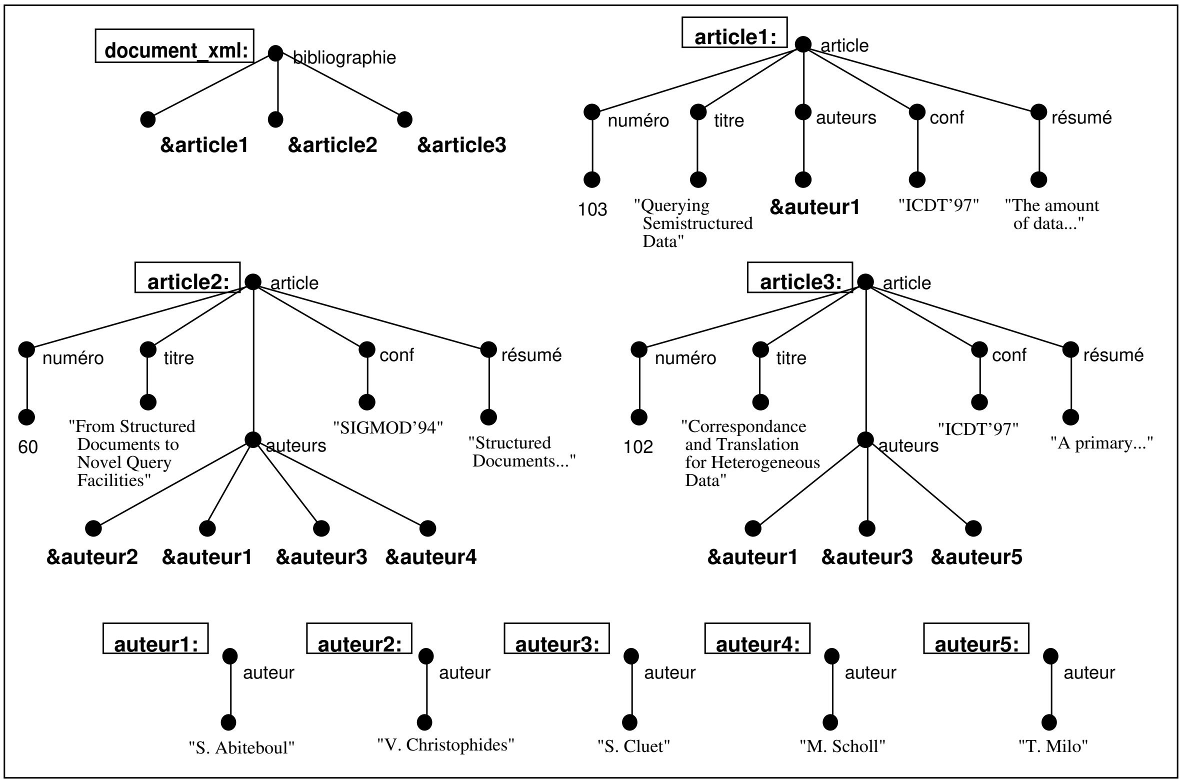


FIG. 4.1: *Un document XML dans le modèle YAT*

la similarité entre le format XML et une structure d'arbre rend la conversion vers le format pivot particulièrement naturelle.

La forêt est l'entité de base du modèle (même si pour des raisons de clarté, la plupart des exemples utiliseront un nombre réduit de motifs). Une forêt est dite valide si elle se compose d'un ensemble *complet* d'arbres nommés. Par complet, nous entendons que tous les noms de motifs utilisés en référence appartiennent bien à la forêt, excluant la possibilité de « pointeurs vides ». Une telle forêt permet de représenter un ensemble cohérent d'informations provenant d'une source telle qu'une base de données, un document, un site sur la Toile, etc. Pour le moment, nous ne décrivons que des forêts *concrètes*, c'est à dire des forêts ne contenant que des valeurs mais comme nous le verrons dans la section 4.1.2, une forêt pourra représenter un schéma ou un méta-modèle. L'ensemble des forêts concrètes, noté $\mathcal{F}_\mathcal{C}$, est le principal ensemble dans lequel nous travaillerons.

Gérer la perte d'information : le choix d'une "bonne" représentation. Regardons maintenant comment le modèle se comporte lors de la représentation d'autres formats. Représenter un objet issu d'une base de données ODMG demande un peu plus d'attention. En effet, le modèle ODMG propose de nombreuses constructions de type (n-uplets, collections homogènes) qui ne peuvent être représentées fidèlement. La figure 4.2 montre un objet ODMG (en haut), et une représentation possible de cet objet (en bas) dans laquelle les valeurs atomiques ainsi que les noms d'attributs et de classes sont utilisés comme annotations.

Notons, tout d'abord, que ce motif ne peut être considéré à lui seul comme une forêt YAT valide, et qu'il est nécessaire de lui adjoindre le motif correspondant à la publication ***p*1**, puis

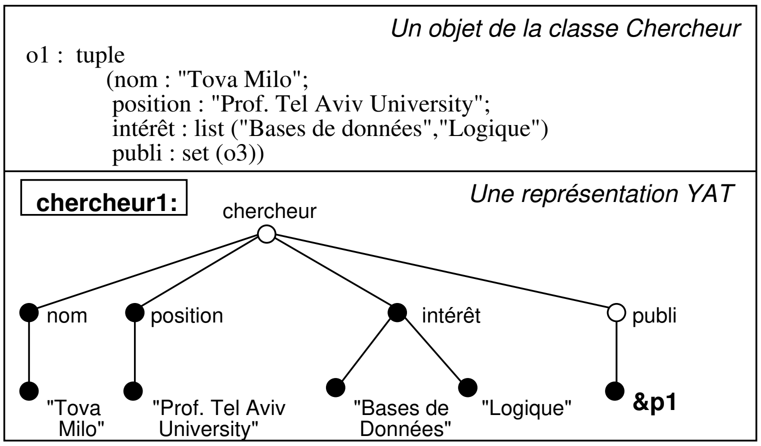


FIG. 4.2: *Représentation d'un objet*

les motifs éventuellement référencés par celui-ci, etc – en tout état de cause, il est naturel de représenter toute la base de données par une unique forêt.

Par ailleurs, remarquons que les noms de motifs permettent de représenter les références entre objets (p.ex. la référence à $o3$ par $\&\boldsymbol{p1}$) de façon naturelle. De plus, l'utilisation à bon escient de nœuds ‘ ○ ’ permet, par exemple, d'indiquer que l'ordre des attributs n'est pas significatif. Rappelons (cf. section 3.1.2) que les formats semistructurés proposés jusqu'ici (à l'exception notable de [BM99]), imposent l'utilisation univoque de collections ordonnées [ACM97] ou non ordonnées [PGW95, BDS95, FFLS97]. Ceci est à l'origine d'un problème délicat si la source offre, comme c'est le cas ici, différents types de collections. Imaginons que l'on représente une collection non ordonnée par une liste, cela entraîne : soit que deux données égales dans le modèle source puissent ne plus l'être dans le modèle pivot, soit que l'adaptateur procède à un réordonnancement, de façon toujours semblable, sur toutes les valeurs de ces types. Cela n'est pas raisonnable, la première solution étant à rejeter pour des raisons sémantiques, la seconde pour d'évidentes raisons de performance.

Malgré tout, la sémantique associée aux constructeurs de type a malheureusement été perdue (p.ex. le fait que les valeurs “Bases de Données” et “Logique” proviennent d'une collection ou que le symbole nom soit un attribut). Ceci est inévitable dans la mesure ou le format pivot ne fournit pas les constructions de type de l'ODMG. En y regardant de plus près, ce choix de représentation possède un autre inconvénient. La figure 4.3 montre deux objets ODMG et deux représentations possibles pour ceux-ci. La première représentation est identique à celle utilisée sur la figure 4.2, tandis que la seconde fournit des annotations additionnelles (tuple, set, etc), indiquant le type d'origine. Remarquons que, dans le premier cas, les deux objets ont des représentations identiques ! On a donc perdu de l'information dans la mesure où il n'est plus possible de revenir à la base de données originelle. Cela est particulièrement dommageable lors des conversions, dans lesquelles on souhaite exporter les données du modèle pivot vers un format cible. Nous préférons donc la seconde représentation qui, d'une part n'entraîne pas de perte d'information (conversion injective) et ajoute à la lisibilité. Ces annotations supplémentaires permettent notamment à l'utilisateur de comprendre quelle était la sémantique de départ et lui donnent également la possibilité de spécifier une intégration qui dépende du type ce qui, nous le verrons, peut être très utile.

Bien entendu, le choix d'une représentation pour un format donné reste en grande partie

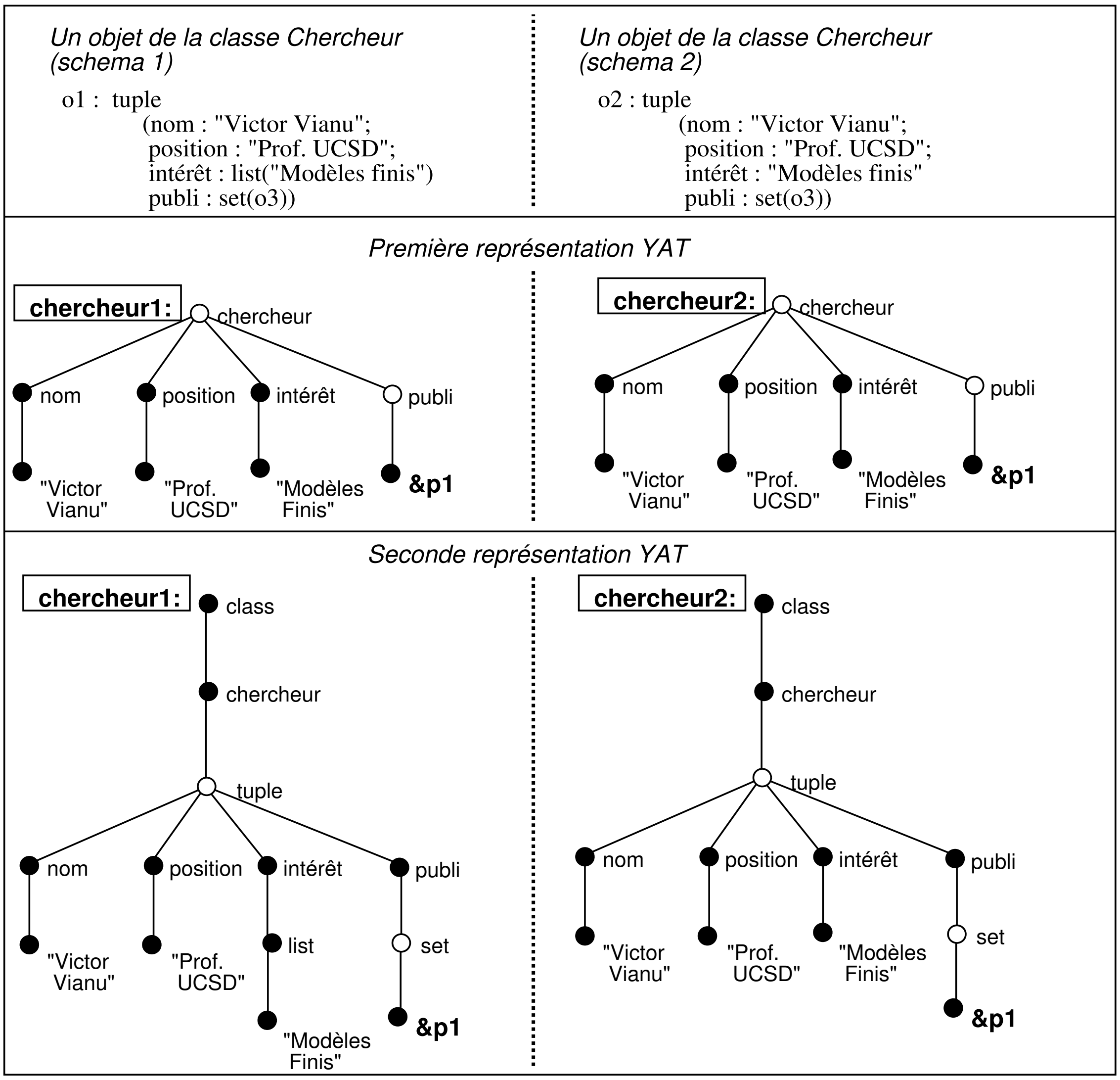


FIG. 4.3: *Deux représentations pour les objets*

un problème de convention. Ceci dit, la conservation de l'information de départ ainsi que la lisibilité de la représentation pivot sont les critères essentiels permettant de distinguer une bonne représentation d'une mauvaise. On peut finalement remarquer que, le modèle YAT étant constitué d'arbres, il est judicieux pour concevoir la représentation d'un format, de s'inspirer de l'arbre de syntaxe abstraite : c'est implicitement ce que nous avions fait pour le document XML de la figure 4.1.

4.1.2 Vers plus d'abstraction

Des types pourquoi faire ? Nous avons déjà souligné l'utilité des informations relatives à la structure des données pour différents usages. La structure est un support sémantique important et une information essentielle pour l'optimisation. Il paraît donc naturel de rajouter du typage à notre modèle pivot. Cependant, il s'agit de ne pas retrouver par la même occasion les inconvénients des approches structurées. Il faut donc être capable de rajouter du contrôle tout en conservant au maximum la souplesse du modèle semistructuré. Nous décrivons ici les constructions de type du modèle YAT, nous verrons en quoi l'approche originale choisie permet d'offrir à la fois contrôle et flexibilité, ou plus précisément de jouer de façon fine sur ces deux paramètres.

Nous souhaitons donc fournir un système de types pour le modèle semistructuré. C'est à dire que nous voulons être capable de décrire des sous-ensembles de l'ensemble des graphes semistructurés que nous avons défini dans la section précédente (c.-à-d. l'ensemble $\mathcal{F}_\mathcal{C}$ des forêts concrètes) ayant certaines propriétés. Rappelons (cf. section 3.1.3) que les questions qui se posent lors de la conception d'un tel système de types sont : quelles propriétés sur les graphes souhaite-t'on exprimer et comment les types vont-ils être obtenus ?

Dans un contexte d'intégration, la plupart des sources fournissent une structure (p.ex. le schéma d'une base de données, la DTD de documents, une grammaire etc). Cette structure peut être plus ou moins précise, dans certains cas absente ou inaccessible. Quoi qu'il en soit, être capable de la réutiliser quand elle existe offre plusieurs avantages. Tout d'abord la structure est fournie par la source elle-même, ce qui évite d'avoir à la demander à l'utilisateur ou d'avoir à la calculer à partir des données (procédé coûteux et peu fiable si les données sont souvent mises à jour). Ensuite, nous sommes assurés d'avoir le meilleur type possible et donc l'information structurelle la plus précise. L'idéal serait que les types YAT coïncident avec les types d'origine des sources le plus souvent possible. Nous allons donc chercher à transposer un maximum des types utilisés par les sources dans notre propre système de types. Dans un contexte hétérogène, celui-ci devra par ailleurs s'adapter aux différents cas de figure : du structuré au semistructuré, en passant par les cas où l'information de type est partiellement connue.

Premier exemple : description d'un schéma ODMG. La première étape vers la description de structures est la représentation de types atomiques et de collections. Nous introduisons pour cela des annotations relatives aux types des labels sur les nœuds des motifs et des annotations relatives à l'arité des fils sur les arêtes.

La figure 4.4 montre un schéma ODMG composé de deux classes (Chercheur et Publication), ainsi que la représentation correspondante dans le modèle de types YAT. Une fois de plus, il s'agit d'une forêt, avec dans le cas présent un motif pour chaque classe. Par rapport aux motifs de la figure 4.3 qui décrivaient des objets, certains labels ont été remplacés par des annotations de type (par exemple String et Int respectivement sous les nœuds titre et année du motif ***Publication***) et certaines arêtes portent le symbole $\star$ pour indiquer une collection. Incidemment, les références (p.ex. **&*Publication***) sont désormais des références entre classes.

Dans YAT, les annotations peuvent être des formules logiques, écrites à partir des types de base (String, Int, Float, Bool et Symbol) et des valeurs atomiques, puis construites par disjonction, conjonction et négation. Par exemple, (Int), (Bool $\vee$ "Vassilis Christophides"), ou encore ($\neg$15) sont des annotations valides. Ceci permet d'offrir une grande flexibilité dans la description des labels, tout en restant relativement simple. Ces annotations décrivent des domaines de labels : par exemple, le domaine de (Int) est $\mathbb{N}$, celui de (Bool $\vee$ "Vassilis Christophides") est $\{\mathsf{true}, \mathsf{false}\} \cup \{$"Vassilis Christophides"$\}$, etc.

Les annotations des arêtes sont constituées d'une borne inférieure (un entier naturel) et d'une borne supérieure (un entier naturel non nul et pouvant être infini). Par exemple $(4, 7)$

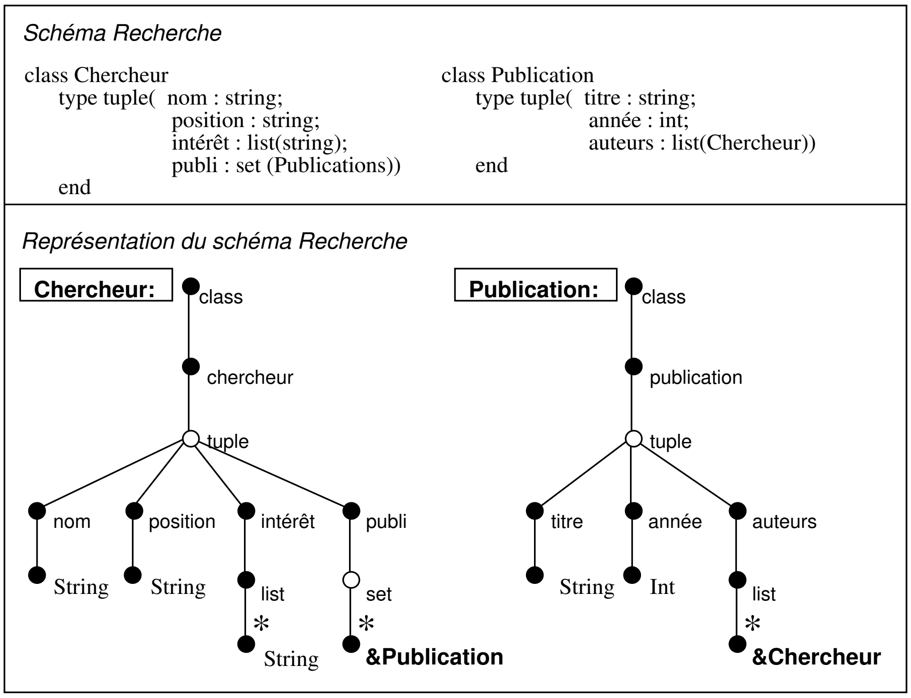


FIG. 4.4: *Représentation d'un schéma ODMG*

indique que le nœud peut avoir de 4 à 7 fils. Dans notre exemple, le symbole $\star$ signifie $(0, \infty)$ et indique une collection avec un nombre quelconque de fils.

Expliquons maintenant dans quelle mesure le motif $\boldsymbol{Chercheur}$ représente bien la classe Chercheur, ou plus précisément, comment le domaine d'une forêt de type est construit. En fait, un motif de type décrit un ensemble de motifs concrets tels que : partant du motif de type, un nœud portant un label est remplacé par un nœud portant une valeur dans le domaine du label (p.ex. String $\mapsto$ "Victor Vianu", nom $\mapsto$ nom), une arête portant l'annotation (n, m) est remplacée par un ensemble d'arêtes $(1, 1)$ dont le nombre est compris entre n et m, et une référence vers un motif est remplacée par une référence à un motif du type correspondant (p.ex. $\boldsymbol{\&Publication} \mapsto \boldsymbol{\&p1}$). Suivant ces règles simples, le motif $\boldsymbol{chercheur1}$ de la figure 4.3 est bien dans le domaine de $\boldsymbol{Chercheur}$ (modulo le fait que $\boldsymbol{p1}$ soit bien de type $\boldsymbol{Publication}$). Cette notion de domaine sera décrite plus précisément par la notion d'instanciation dans la section suivante.

On peut d'ores et déjà se poser la question de la précision du typage obtenu par rapport au type d'origine des données. C'est à dire, la forêt correspondant au schéma ODMG Recherche décrit-elle précisément l'ensemble des instances de ce schéma ? On peut aisément vérifier l'inclusion dans les deux sens. Toutes les bases de données, instances du schéma Recherche sont bien représentées par des instances de la forêt YAT Recherche, et réciproquement. Cela signifie donc que la forêt de la figure 4.4 représente précisément le schéma ODMG dont elle est issue.

Plus de généricité : le modèle ODMG. Nous allons maintenant essayer de capturer l'ensemble des graphes valides par rapport au modèle ODMG. Pour permettre des descriptions encore plus génériques, nous rajoutons l'union de type (notée $\vee$) et une seconde notion de

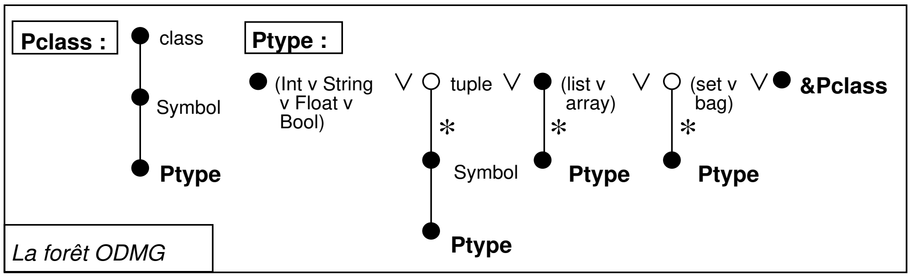


FIG. 4.5: *Vers une représentation complète des graphes ODMG*

référence permettant de distinguer entre références par identifiants (notées ***&ref***) ou par valeurs (notées ***ref***). L'union de type sert à décrire plusieurs alternatives de type possibles et la référence par valeur va nous permettre de décrire des structures récursives.

La figure 4.5 exploite ces deux nouvelles constructions pour représenter le méta-modèle ODMG. Seuls deux motifs sont utilisés. Le motif ***Pclass*** indique que les objets ODMG commencent par un nœud class, suivi du nom de la classe de type Symbol, suivi d'un motif (qui est instance de ***Ptype*** mais pas une référence à ***Ptype***), décrivant la valeur de l'objet. Le motif ***Ptype*** décrit les valeurs ODMG comme l'union des cas suivants : soit une valeur atomique (Int ou String, etc), soit un n-uplet, soit une collection, soit la référence à un objet. Remarquez que le motif ***Ptype*** décrit un type récursif. Nous voyons ainsi comment représenter une information encore plus abstraite qu'un schéma.

Cependant, et sans rentrer dans les détails du mécanisme d'instanciation, cet exemple illustre aussi les limites du modèle YAT. En effet, la forêt ODMG donnée sur la figure 4.5 n'a pas la bonne propriété qu'avait la forêt Recherche de la figure 4.4. En effet, une base ODMG sera bien toujours représentée par une instance de la forêt ODMG, mais la réciproque est fausse. Ceci est dû au fait que cette forêt ODMG n'est pas suffisamment précise pour décrire exactement l'ensemble des forêts concrètes ODMG. En particulier, elle ne décrit ni le fait que les noms d'attributs doivent être distincts, ni que les collections doivent être homogènes. Par exemple set(15, tuple (a: "Vassilis Christophides")) n'est certainement pas une valeur ODMG mais le motif correspondant est bien une instance de la forêt ODMG. Insistons cependant sur le fait que malgré ces lacunes, la forêt ODMG, telle qu'elle est fournie, donne des informations intéressantes à l'utilisateur : elle peut servir de guide lors de l'écriture de programmes d'intégration génériques pour les bases ODMG ou encore être utilisée lors de l'optimisation.

Dorénavant, un motif YAT sera une union de motifs dont les arêtes portent des annotations d'arité, dont les nœuds portent des annotations de type, et dont les feuilles peuvent porter des références. Une forêt YAT valide sera, une fois encore, un ensemble complet de motifs. Le modèle YAT est composé de l'ensemble, noté $\mathcal{F}$, de toutes les forêts valides. Enfin, il est important de remarquer que l'ensemble $\mathcal{F}_{\mathcal{C}}$ des forêts concrètes est inclus dans $\mathcal{F}$. En effet les motif concrets sont tous les motifs sans union, dont les arêtes sont annotées par $(1,1)$, dont les nœuds sont annotés par des valeurs atomiques et n'ayant aucune référence par valeur.

Précision vs généricité. Nous avons maintenant toutes les constructions permettant de décrire des informations abstraites. En particulier, nous avons vu comment représenter, dans le cadre du modèle ODMG plusieurs niveaux d'abstraction. De manière générale, la représentation YAT de modèles structurés se fait souvent en utilisant : (i) des labels pour les valeurs

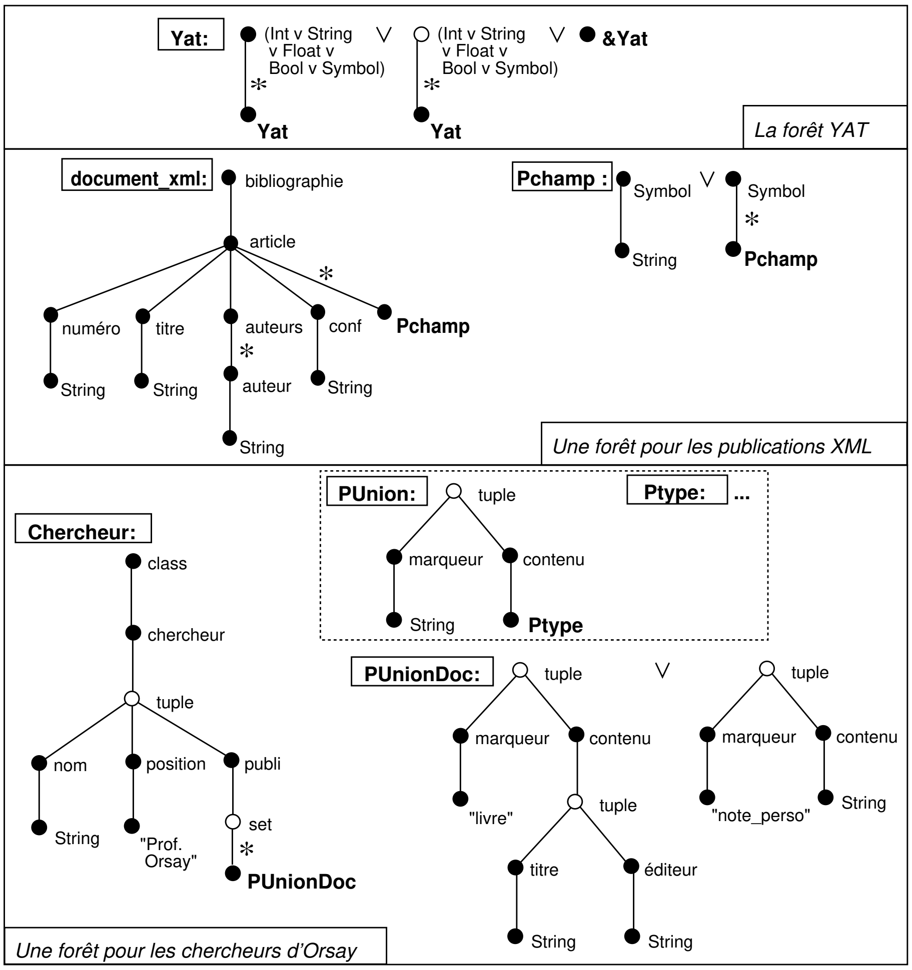


FIG. 4.6: *Trois façons d'exploiter la généricité*

atomiques (e.g. “Vassilis Christophides”), (ii) des labels pour les informations du schéma (e.g. nom, chercheur), et (iii) des labels pour les informations de type (e.g. class, tuple). Le premier niveau d'abstraction (i.e. le schéma) s'obtient en remplaçant les valeurs atomiques par leurs types, et le second (i.e. le méta-modèle) en remplaçant les informations liées au schéma également par leurs types. Notons qu'à chaque fois, l'on gagne en généricité mais l'on perd en précision de la description. Nous retrouverons le pendant de cette propriété lors de la spécification de programmes d'intégration.

Bien que cette façon d'utiliser le typage (*i.e.* méta-modèle $\succ$ schéma $\succ$ instance) ait son intérêt dans un contexte structuré, le modèle YAT n'est pas limité à un tel mode d'utilisation. En fait, il permet de mélanger arbitrairement des descriptions plus ou moins structurées. La figure 4.6 illustre trois utilisations originales, voire curieuses, de la puissance de représentation offerte. La forêt YAT présentée en haut de la figure est particulièrement intéressante car elle décrit toutes les forêts possibles du modèle. Le motif ***Yat*** décrit, en effet, des arbres construits récursivement : soit d'un nœud annoté avec de l'ordre et un nombre quelconque de fils, soit d'un nœud annoté sans ordre et avec un nombre quelconque de fils, soit d'une feuille avec une référence. Il s'agit donc bien d'un type décrivant tous les motifs possibles ! En conséquence, il permet d'être complètement générique mais n'offre par ailleurs aucune possibilité de contrôle.

En quelque sorte, nous avons décrit le typage, ou plutôt l'absence de typage, des modèles semistructurés.

La forêt XML au milieu de la figure 4.6 illustre comment représenter une information structurelle partielle. En effet, celle-ci indique qu'une publication doit toujours avoir un numéro, un titre, un ensemble d'auteurs et un nom de conférence, mais qu'elle peut aussi contenir un nombre quelconque de champs additionnels (motif ***Pchamp***), ces champs étant nommés (nœud Symbol) et pouvant contenir une chaîne de caractères (nœud String) ou des sous-champs imbriqués (nœud ***Pchamp***). Nous voyons qu'il est possible de transmettre au système une connaissance imparfaite sur la structure des données. Celui-ci peut alors s'en servir pour manipuler les informations correspondantes (cf. chapitre 5) et à des fins d'optimisation (cf. chapitre 6). Notons au passage que, à l'inverse de la représentation XML de la figure 4.1, celle-ci ne contient plus de référence : les documents XML stockent le contenu des documents sans utiliser de liens.

Enfin, le bas de la figure 4.6 montre une utilisation originale des capacités du modèle. Dans cet exemple, nous reproduisons la représentation de l'union de type marquée dans le modèle ODMG, telle qu'elle est utilisée dans [Chr96] pour le stockage de documents SGML dans une base de donnée à objets. Le motif ***Publication*** décrit des chercheurs d'Orsay dont l'attribut publication regroupe un ensemble de documents (***PUnionDoc***). Les documents en question peuvent être soit un livre, soit une note publiée à titre personnel ce qui pourrait correspondre au fragment de DTD suivant :

```
<! ELEMENT publi        - - (livre | note_perso) >
<! ELEMENT livre        - - (titre,éditeur) >
<! ELEMENT titre        - - (#PCDATA) >
<! ELEMENT éditeur      - - (#PCDATA) >
<! ELEMENT note_perso   - - (#PCDATA) >
```

L'union est représentée par un ensemble de n-uplets, chacun avec un marqueur (indiquant de quelle alternative il s'agit) et un contenu (qui dépend du marqueur !). Les motifs entre pointillés en haut à gauche décrivent la structure d'union de type la plus générale possible. Cela serait l'information la plus précise fournie à l'utilisateur dans [Chr96]. En mélangeant informations spécifiques et génériques, le motif ***PUnionDoc*** donne à l'utilisateur une information plus précise quand à la structure des données stockées dans la base.

Cette série d'exemples démontre la flexibilité du modèle et son potentiel pour la description d'une connaissance structurelle incomplète.

4.1.3 Instanciation et typage

Nous avons plusieurs fois évoqué la notion d'instanciation lors de l'interprétation des forêts YAT. L'instanciation permet de décrire le lien entre une forêt plus générique et une autre plus spécifique. Il s'agit donc du mécanisme de base nous permettant de jouer de façon très fine sur le degré de généricité. Nous commençons par décrire l'instanciation puis montrons comment ce mécanisme peut être utilisé pour le typage.

Le mécanisme d'instanciation. Deux forêts sont instances l'une de l'autre si une relation existe entre les nœuds et arêtes de ces forêts, telle que les labels des nœuds de l'instance ont

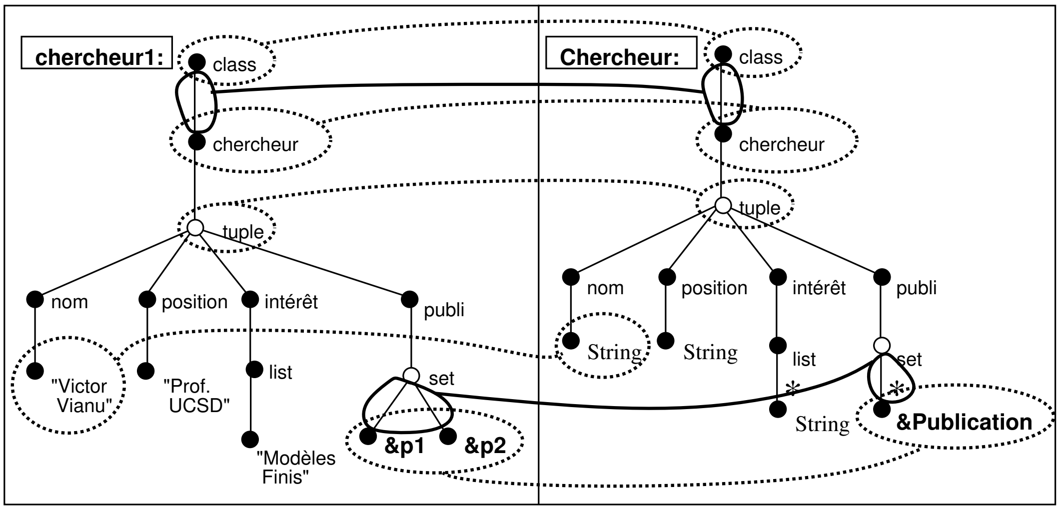


FIG. 4.7: *Un exemple d'instanciation*

un domaine plus petit (p.ex. $\mathsf{Int} \mapsto 15$) et telle que le nombre d'arêtes de l'instance est inclus dans l'intervalle indiqué par l'arête qui lui correspond (p.ex. quatre fils pour un intervalle $(2, 5)$). Par exemple, le motif ***chercheur1*** (figure 4.3) est instance du motif ***Chercheur*** (figure 4.4), qui lui même, est une instance de ***Pclass*** (figure 4.5). De même, tous les motifs que l'on a rencontrés sont instances du motif ***Yat*** (figure 4.6).

Plus précisément, une forêt F' est une instance d'une forêt F si et seulement si, il existe une fonction *map*, des nœuds et arêtes de F' vers des sous-ensembles non vides de nœuds et arêtes de F, qui vérifie les propriétés suivantes.

- Si n est un nœud de F' alors les nœuds de $map(n)$ sont de même nature (labels ou références, ordonnés ou non ordonnés, partageables ou non partageables).

- Si n est un nœud de F' avec un label, celui-ci est *plus petit* (par rapport à l'ordre partiel induit par l'inclusion des domaines associés) que celui de $map(n)$ (p.ex. $\mathsf{Int} \geq 15$).

- Si n est un nœud de F' avec une référence, alors celle-ci est une instance de la référence de $map(n)$.

- Si $a_1, \ldots, a_n$ sont des arêtes de F' associées à une même arête de F, alors l'ensemble de leurs annotations est *plus petit* que celui de l'arête qui leur est associée (plus petit par rapport à un ordre partiel sur les annotations d'arêtes qui préserve ou restreint l'intervalle spécifié).

La figure 4.7 illustre ce mécanisme. Les traits pointillés décrivent partiellement (pour des raisons de lisibilité) la fonction qui met en relation les nœuds du motif ***Chercheur*** et de son instance ***chercheur1*** : par exemple "Victor Vianu" $\leq$ String, class $\leq$ class. Les traits gras décrivent partiellement la relation entre les arêtes. Par exemple, celle concernant l'annotation $\star$ est valide car $\{(1,1),(1,1)\} \leq \{(0,\infty)\}$. Notons que l'instanciation est une relation sur les forêts, qui n'est donc vérifiée dans notre exemple que dans la mesure où ***p1*** et ***p2*** sont bien des instances de ***Publication***.

Utilisation pour le typage. La relation d'instanciation est définie de manière générale sur toutes les forêts du modèle. Les forêts concrètes sont plongées dans le système de type : on peut en déduire deux interprétations distinctes relatives au typage. Premièrement, l'instanciation d'une forêt en une forêt concrète montre que celle-ci est typée par la première (p.ex. ***chercheur*1** est de type ***Chercheur***). Deuxièmement, l'instanciation entre deux forêts quelconques capture une forme de sous-typage, appelée subsomption dans [BDFS97] (p.ex. ***Chercheur*** subsume ***Pclass***). Plus précisément :

1. **typage**: Soit F une forêt et F' une forêt concrète. On dira que F' est de type F si F' est une instance de F. Ou de façon équivalente, le *domaine* de F (noté $Dom(F)$) est l'ensemble des forêts concrètes F' instances de F.

2. **subsomption**: Une forêt F' *subsume* une forêt F si elle lui est instance. Intuitivement, ceci est lié au fait que le domaine d'une instance de F est toujours inclus dans le domaine de F.

Notons que l'on peut même considérer une forêt concrète comme un type, le domaine étant alors un singleton contenant ce même graphe.

4.2 Filtrage

Le filtrage est le mécanisme que nous utiliserons pour extraire des données à partir d'un graphe semistructuré. Nous avons choisi d'utiliser un tel mécanisme en raison de sa nature déclarative, de sa concision, et de sa sémantique relativement intuitive. Le filtrage est utilisé de façon intensive dans le langage d'intégration qui sera décrit au chapitre 5, mais nous avons préféré l'introduire dès maintenant car il se fonde sur la notion d'instanciation que nous venons de présenter. Également, une conséquence inattendue [1] mais particulièrement intéressante de la conception des filtres sera la facilité avec laquelle nous typerons le langage (cf. section 5.2).

4.2.1 Filtres d'entrée

Introduction. L'objectif des filtres est l'extraction d'informations à partir d'un motif ou d'une forêt YAT. Nous avons choisi de nous inspirer de [ACM97] qui utilise un langage logique à la datalog. Les règles du langage s'appliquent par *unification* : c'est à dire que lorsqu'un fait d'entrée s'apparie avec un prédicat du langage, il en découle la liaison des variables contenues dans le prédicat. Dans [ACM97], les prédicats servent à décrire des structures d'arbre et jouent donc le rôle de filtres.

Du fait de l'existence d'un système de type pour YAT, nous avons choisi d'étendre les filtres présents dans [ACM97] pour qu'ils coïncident avec les expressions du système de types. Le système par filtrage qui en résulte offre de nombreuses similitudes avec celui utilisé dans les langages de programmation fonctionnels [Que90, Mar92], notamment il est trivial à typer.

En un mot, les filtres YAT sont des motifs quelconques auxquels on attache des variables. Les filtres sont appliqués sur des motifs concrets en deux étapes.

1° On vérifie que le motif concret est bien une instance du motif sous-jacent au filtre.

1. À moitié, il va sans dire !

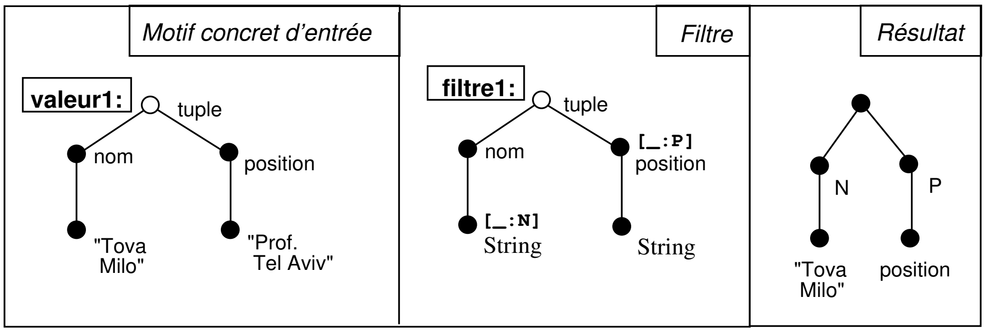


FIG. 4.8: *Un filtrage élémentaire*

2° Si c'est le cas, la correspondance entre les nœuds et arêtes du graphe (voir la fonction *map* de la section précédente) permet de déduire un ensemble de liaisons pour les variables du filtre.

La figure 4.8 illustre cette idée : nous tentons de filtrer le motif ***valeur*1** à travers ***filtre*1**. Celui-ci présente deux variables, notées **N** et **P**. Premièrement, ***valeur*1** est bien une instance de ***filtre*1**. Deuxièmement, on en déduit les liaisons suivantes entre les variables et les labels : **N** $\mapsto$ "Tova Milo" et **P** $\mapsto$ position. Ces liaisons sont représentées sur la figure par un motif dont chaque fils correspond à un couple (variable, valeur).

Que peut-on lier ? Il est possible d'écrire des filtres plus complexes, les choses se compliquent alors quelque peu. Tout d'abord, dès qu'un filtre utilise des annotations pour les collections, une variable peut être liée à un ensemble de valeur plutôt qu'à une seule. Pour capturer cela, nous choisissons une sémantique de produit cartésien sur les ensembles de liaisons. Ensuite, les labels ne sont pas les seules informations que l'on peut avoir envie d'extraire, il est donc important de fournir plusieurs sortes de variables.

La figure 4.9 illustre ces deux aspects sur l'exemple d'instanciation de la figure 4.7. On peut tout d'abord remarquer que la variable **Pub** est liée à deux nœuds distincts. C'est pourquoi le résultat du filtrage est décrit par un motif avec deux fils, un pour chaque groupe de liaisons obtenues après produit cartésien. Ensuite, on peut noter l'utilisation de variables simultanément sur les nœuds et sur les arêtes :

- Sur un nœud on peut lier le label (p.ex. **N**) ou le motif lui même (c'est à dire tout l'arbre, comme pour **PL**). Pour distinguer les deux cas, la notation [**v1:v2**] indique que **v1** est une variable de motif, tandis que **v2** est une variable de label. Le symbole '_' indique l'absence de variable. Par ailleurs, dans le cas où le label est une référence, on souhaite également pouvoir distinguer les variables liées à un tel label, des variables liées au motif associé à cette référence. Par conséquent, les variables de label préfixées par le symbole & indiqueront une déréférenciation. Contrairement à ce qu'il pourrait sembler, nous suivons bien en cela les conventions habituelles. En effet, pour la variable [_:&**Pub**] de notre filtre d'exemple, &**Pub** est liée aux références &***p*1** et &***p*2** et donc **Pub** est liée aux motifs (valeurs) ***p*1** et ***p*2**.

- Sur une arête, on peut lier l'annotation d'arité (p.ex. **I**) ou la collection qui correspond

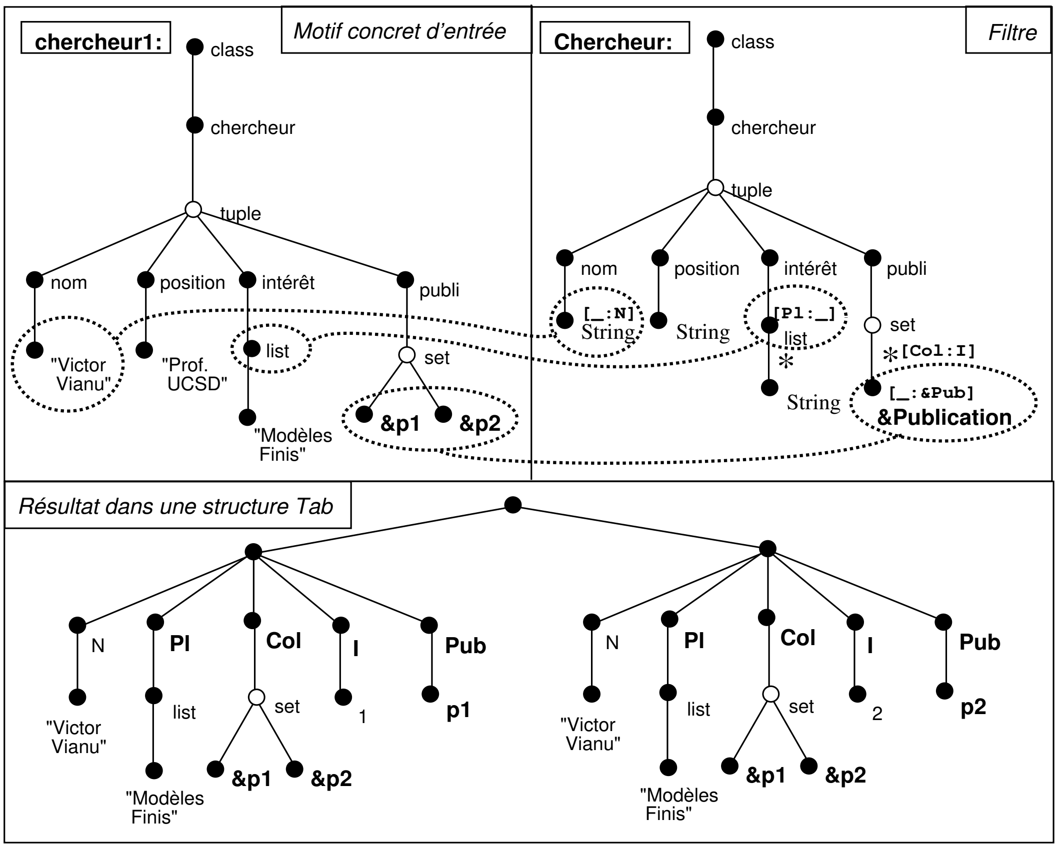


FIG. 4.9: *Un filtrage avec variables d'arêtes et de nœuds*

à l'arête, c'est à dire le sous-arbre correspondant à l'arête[2] (p.ex. **Col**). Pour distinguer les deux cas, la notation [**v3:v4**] indique que **v3** est une variable de motif, tandis que **v4** est une variable d'arité.

Notons que l'utilisation d'une sémantique de produit cartésien entraîne que la structure utilisée pour la représentation des liaisons est régulière. Elle est similaire à une relation qui n'est pas en première forme normale, avec les noms des variables comme attributs. Ces structures, que nous serons amenés à rencontrer de nouveau dans le chapitre 6, s'appellent des Tab[3].

Produit cartésien et contexte. Pour terminer cette présentation du filtrage, regardons comment se passent les choses sur des structures plus imbriquées. La figure 4.10 montre le filtrage d'un ensemble de n-uplets dont l'un des attributs est lui-même un ensemble.

Ce filtrage nous amène à faire deux remarques. Tout d'abord, la liaison permet de conserver la relation qui existe entre un auteur et ses publications. Ceci est dû au fait que les variables **Pub** et **N** sont issues du même nœud (annoté par tuple). Ensuite, la sémantique de produit cartésien implique qu'aucune information sur "Gaston" n'est conservée.

2. Bien que cela n'apparaisse pas dans l'exemple, celui-ci est différent du motif complet lorsqu'il y a plusieurs arêtes.

3. Pour tableaux à deux dimensions.

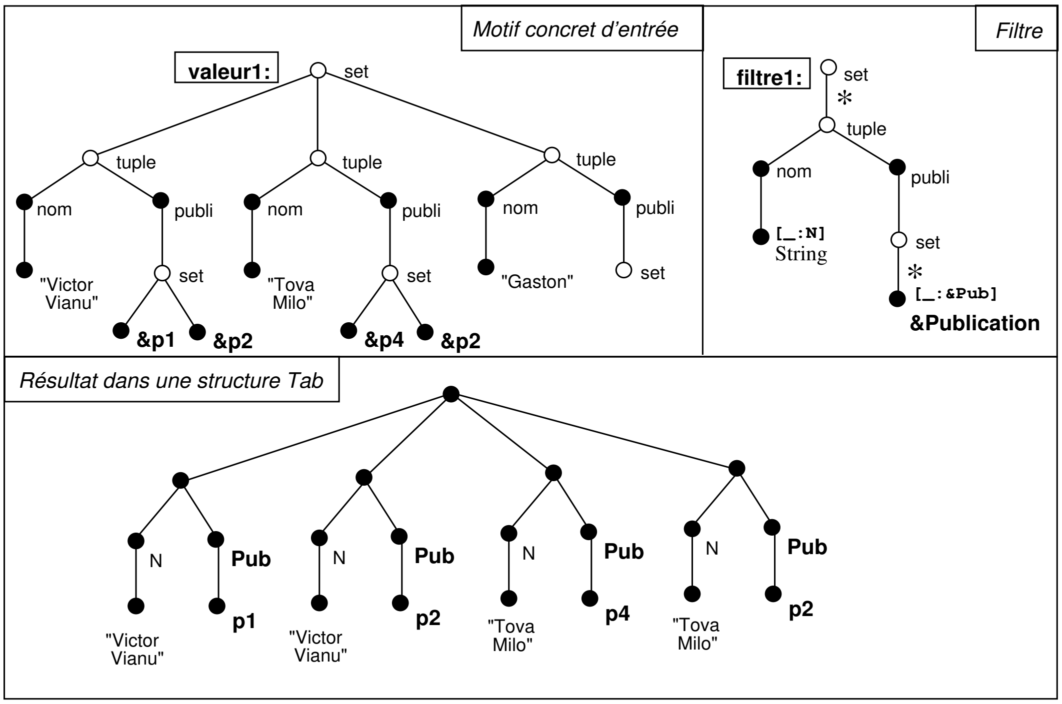


FIG. 4.10: *Filtrage de collections imbriquées*

4.2.2 Filtres de sortie

Les filtres de sortie jouent un rôle exactement complémentaire à celui des filtres d'entrée. Partant d'une structure de Tab, ils décrivent comment construire un motif YAT en sortie. On pourrait imaginer que ceux-ci soient exactement les mêmes que les filtres d'entrée. Malheureusement, cela conduit à des ambiguïtés dans de nombreux cas. En effet, partant d'un même filtre et d'une même structure de Tab, il existe souvent plusieurs motifs concrets qui peuvent convenir. Partant de l'exemple précédent, la figure 4.11 illustre une telle situation. Les deux valeurs ***valeur*1** et ***valeur*2** sont toutes les deux des valeurs possibles pour le filtre et la structure Tab fournis en entrée.

Pour lever ces ambiguïtés, les filtres de sortie fourniront des primitives de regroupement et d'ordonnancement : ceci sera illustré en détail dans le prochain chapitre. Notons pour finir que lors d'une telle opération, des contraintes quant aux types des variables et quant à leur position doivent être respectées. Il est évident que les types des variables dans le Tab et dans le filtre de sortie doivent coïncider et qu'une variable contenant un motif ne peut être utilisée comme label !

4.3 Une formalisation

Nous donnons une formalisation du modèle de données, ainsi qu'une définition complète de la relation d'instanciation. Nous ne nous lancerons pas dans une étude théorique détaillée du modèle mais l'objectif premier de cette formalisation est de servir de support à une implantation rigoureuse du modèle dans le système YAT.

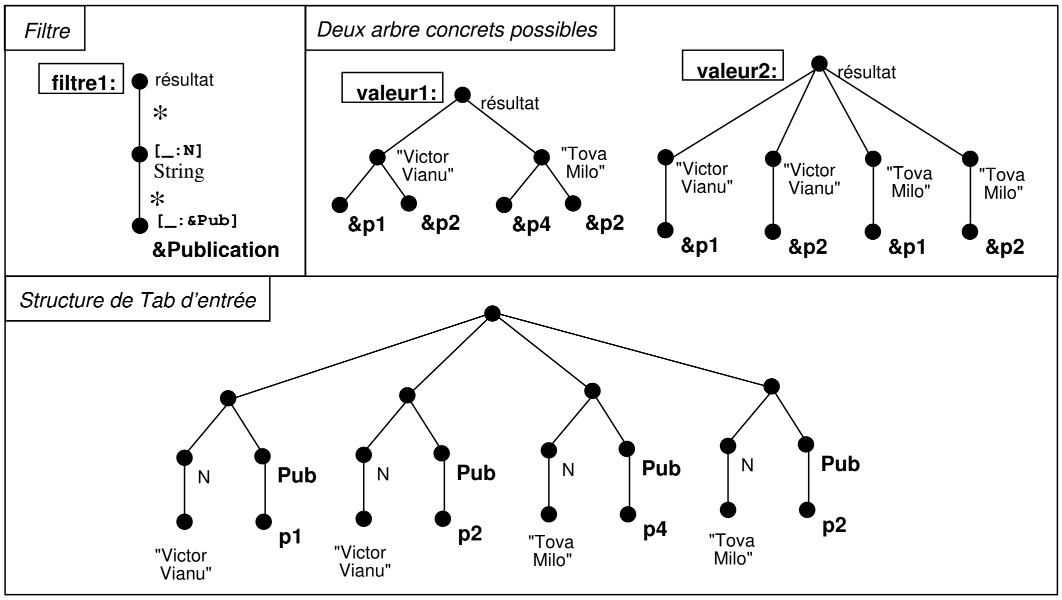


FIG. 4.11: *En sens inverse...*

4.3.1 Motifs

Symboles de base. Nous supposons l'existence des ensembles infinis et disjoints suivants :

1° $\mathbf{T} = \{\mathsf{Bool}, \mathsf{Int}, \mathsf{String}, \mathsf{Float}, \mathsf{Symbol}\}$ l'ensemble des types atomiques ;

2° $\mathbf{C}$ un ensemble de constantes appartenant aux types $\mathbf{T}$, et *dom* la fonction décrivant les domaines dans $\mathbf{C}$ des types atomiques de $\mathbf{T}$;

3° $\mathbf{N} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \dots\}$ un ensemble de noms de motifs ;

4° $\mathbf{E}$ un ensemble de couple d'entiers, utilisés comme symboles d'arité, tel que : $\mathbf{E} = \{(n, m) \in \mathbb{N} \times (\mathbb{N} \cup \{+\infty\}) \mid n \leq m\}$.

Nous supposons également l'existence d'un symbole particulier, noté '_'. Ce symbole jouera dans toute la suite le rôle d'une annotation vide, tant pour les labels que pour les variables.

Tout ce que nous dirons dans la suite s'applique à des types atomiques ayant des domaines non vides et disjoints. Nous supposerons donc que les domaines des types de $\mathbf{T}$ forment une partition de $\mathbf{C}$.

Annotations de type.

Définition 4.3.1 *Annotations de type.* L'ensemble $\mathcal{L}$ des annotations de types est défini de la façon suivante :

- $\mathbf{T} \subset \mathcal{L}$
- $\mathbf{C} \subset \mathcal{L}$
- $l_1 \vee l_2$, $l_1 \wedge l_2$, et $\neg l_1 \in \mathcal{L}$ pour tout $(l_1, l_2) \in \mathcal{L}^2$.

□

Définition 4.3.2 *Domaine des annotations de type.* La fonction de domaine *domt* sur les annotations de type est construite à partir de la fonction *dom* de la façon suivante :

- si $a \in \mathbf{T}$, alors $domt(a) = dom(a)$.
- si $c \in \mathbf{C}$, alors $domt(c) = \{c\}$.
- si $(l_1, l_2) \in \mathcal{L}^2$ alors $domt(l_1 \vee l_2) = domt(l_1) \cup domt(l_2)$, $domt(l_1 \wedge l_2) = domt(l_1) \cap domt(l_2)$ et $domt(\neg l_1) = \mathbf{C} \setminus domt(l_1)$ (hypothèse du monde clos).

□

Nous dirons qu'une annotation de type est concrète si son domaine est réduit à un singleton. L'instanciation utilisera l'ordre partiel sur les annotations de type induit par l'inclusion des domaines. Enfin, notons que tout ce que nous dirons s'étend trivialement à toute famille de langage portant sur les constantes de $\mathbf{C}$ et fermé par application des opérations logiques élémentaires (conjonction, disjonction et négation).

Ordre sur les annotations d'arité. Dans la suite, nous utiliserons l'ordre suivant sur l'ensemble $\mathbf{E}$ d'annotations d'arêtes :
$\{(n_1, m_1), \ldots, (n_i, m_i)\} \preceq \{(n'_1, m'_1), \ldots, (n'_j, m'_j)\}$ si et seulement si, $\Sigma_{k=1}^{i} n_k \geq \Sigma_{k=1}^{j} n'_k$ et $\Sigma_{k=1}^{i} m_k \leq \Sigma_{k=1}^{j} m'_k$.

Motifs.

Définition 4.3.3 *Valeurs de motifs.* L'ensemble $\mathcal{PT}$ de valeurs de motifs est défini de la façon suivante :

- si $l \in \mathcal{L}$, alors $l \in \mathcal{PT}$.
- si $\boldsymbol{p} \in \mathbf{N}$, alors $\boldsymbol{p} \in \mathcal{PT}$ et $\&\boldsymbol{p} \in \mathcal{PT}$.
- si $n \in \mathbb{N}^+, l \in \mathcal{L}, (t_1, \ldots, t_n) \in \mathcal{PT}^n, (\epsilon_1, \ldots, \epsilon_n) \in \mathbf{E}^n$, alors $l\langle \stackrel{\epsilon_1}{\rightarrow} t_1, \ldots, \stackrel{\epsilon_n}{\rightarrow} t_n\rangle \in \mathcal{PT}$, et $l\{\stackrel{\epsilon_1}{\rightarrow} t_1, \ldots, \stackrel{\epsilon_n}{\rightarrow} t_n\} \in \mathcal{PT}$.

□

soit t une valeur de motif, nous noterons $ref(t)$ l'ensemble des références (par valeur ou par identifiant) dans t.

Définition 4.3.4 *Motifs.* soit $\boldsymbol{p}$ un nom de motif dans $\mathbf{N}$ et $(t_1, \ldots, t_n)$ un ensemble fini non vide de valeurs de motifs dans $\mathcal{PT}$, alors $\boldsymbol{p} : t_1 \vee t_2 \vee \ldots \vee t_n$ est un motif. Nous appellerons $\mathcal{P}$ l'ensemble des motifs. □

Forêts.

Définition 4.3.5 *Forêts.* Une forêt F est un ensemble de motifs. □

Une forêt F est complète si tous les noms de motifs qui sont utilisés comme références dans F, appartiennent à F. Plus formellement, F est complète si et seulement si, $\forall \boldsymbol{p} \in F, ref(\boldsymbol{p}) \subseteq F$. Nous appellerons $\mathcal{F}$ l'ensemble des forêts complètes.

Une forêt F est dite concrète, si et seulement si, tous les motifs de F sont concrets. L'ensemble de toutes les forêts concrètes et complètes est noté $\mathcal{F}_{\mathcal{C}}$.

4.3.2 Mécanisme d'instanciation

Fonction d'assignation de noms de motifs. Une fonction d'assignation de noms de motifs décrit une correspondance entre noms de motifs.

Définition 4.3.6 *Assignation de noms de motifs.* Une fonction d'assignation de noms de motifs π, est une fonction de $\mathbf{N}$ dans l'ensemble $\mathcal{P}(\mathbf{N})$ des parties de $\mathbf{N}$. Une telle fonction sera dite disjointe, si et seulement si, elle met en correspondance les noms de motifs à des sous-ensembles disjoints de noms de motifs. Plus formellement : $\forall(\boldsymbol{p},\boldsymbol{p'}) \in \mathcal{P}(\mathbf{N}), \boldsymbol{p} \neq \boldsymbol{p'} \Rightarrow \pi(\boldsymbol{p}) \cap \pi(\boldsymbol{p'}) = \emptyset$ □

Instanciation de valeurs de motif. Une valeur de motif peut être instanciée par rapport à la forêt à laquelle elle appartient (en raison de l'utilisation possible de références). L'instanciation de valeurs de motifs est définie récursivement de la façon suivante :

Définition 4.3.7 *Instanciation de valeurs de motif.* Soit F une forêt complète, π une fonction d'assignation disjointe, t et t' deux valeurs de motif. t' est une instance de t dans F, si et seulement si, l'une des propriétés suivantes est vraie :

1. $t = l$ et $t' = l'$ et $domt(l') \subseteq domt(l)$;
2. $t = \&\boldsymbol{p}$, $t' = \&\boldsymbol{p'}$ et $\boldsymbol{p'} \in \pi(\boldsymbol{p})$;
3. $t = \boldsymbol{p}$, $t' = \boldsymbol{p'}$ et $\boldsymbol{p'} \in \pi(\boldsymbol{p})$;
4. $t = \boldsymbol{p}$, et $\boldsymbol{p} : t_1 \vee \ldots \vee t_n \in F$, et t' est une instance de l'un des t_i ;
5. $t = l\langle \stackrel{\epsilon}{\rightarrow} t_1\rangle$, $t' = l'\langle \stackrel{\epsilon'_1}{\rightarrow} t'_1, \ldots, \stackrel{\epsilon'_k}{\rightarrow} t'_k\rangle$, $domt(l') \subseteq domt(l)$, $\{\epsilon\} \preceq \{\epsilon'_1, \ldots, \epsilon'_k\}$, et $\forall i, t'_i$ est une instance de t_1 dans F ;
6. $t = l\langle \stackrel{\epsilon_1}{\rightarrow} t_1, \ldots, \stackrel{\epsilon_n}{\rightarrow} t_n\rangle$, $t' = l'\langle \stackrel{\epsilon_{1,1}}{\rightarrow} t_{1,1}, \stackrel{\epsilon_{1,2}}{\rightarrow} t_{1,2}, \ldots, \stackrel{\epsilon_{1,m_1}}{\rightarrow} t_{1,m_1}, \ldots, \stackrel{\epsilon_{n,m_n}}{\rightarrow} t_{n,m_n}\rangle$, et $\forall i \in [1..n]$ $l'\langle \stackrel{\epsilon_{i,1}}{\rightarrow} t_{i,1}, \stackrel{\epsilon_{i,2}}{\rightarrow} t_{i,2}, \ldots, \stackrel{\epsilon_{i,m_i}}{\rightarrow} t_{i,m_i}\rangle$ est une instance de $l\langle \stackrel{\epsilon_i}{\rightarrow} t_i\rangle$ dans F.
7. $t = l\{\stackrel{\epsilon_1}{\rightarrow} t_1, \ldots, \stackrel{\epsilon_n}{\rightarrow} t_n\}$, $t' = l'\{\stackrel{\epsilon'_1}{\rightarrow} t'_1, \stackrel{\epsilon'_2}{\rightarrow} t'_2, \ldots, \stackrel{\epsilon'_m}{\rightarrow} t'_m\}$, et il existe une permutation σ de $[1..m]$, telle que $l'\langle \stackrel{\epsilon'_{\sigma(1)}}{\rightarrow} t'_{\sigma(1)}, \stackrel{\epsilon'_{\sigma(2)}}{\rightarrow} t'_{\sigma(2)}, \ldots, \stackrel{\epsilon'_{\sigma(m)}}{\rightarrow} t'_{\sigma(m)}\rangle$ est une instance de $t = l\langle \stackrel{\epsilon_1}{\rightarrow} t_1, \ldots, \stackrel{\epsilon_n}{\rightarrow} t_n\rangle$ dans F.

□

Instanciation de motifs.

Définition 4.3.8 *Instanciation de motifs.* Soit F une forêt complète de $\mathcal{F}$, π une fonction d'assignation disjointe et $\boldsymbol{p} : t_1 \vee \ldots \vee t_n$ un motif de F. Un motif $\boldsymbol{p'} : t'_1 \vee \ldots \vee t'_{n'}$ est une instance de $\boldsymbol{p}$ dans F, si et seulement si, $\boldsymbol{p'} \in \pi(\boldsymbol{p})$ et $\forall i \in [1..n'], \exists j \in [1..n]$ tel que t'_i est une instance de t_j dans F. □

Instanciation de forêt et domaine.

Définition 4.3.9 *Instanciation de forêt.* Soient F et F' deux forêts complètes de $\mathcal{F}$. F' est une instance de F, si et seulement si, il existe une fonction d'assignation disjointe π, telle que pour tout $\boldsymbol{p'}$ dans F', il existe un $\boldsymbol{p}$ de F et $\boldsymbol{p'}$ est une instance de $\boldsymbol{p}$ dans F. □

Le domaine de F, noté $Dom(F)$, est l'ensemble de toutes les forêts complètes instances de F. Le domaine concret de F, noté $Dom_C(F)$ est l'ensemble de toutes les forêts complètes et concrètes instances de F.

4.3.3 Filtres

Annotations de variables. Nous supposons l'existence d'un ensemble supplémentaire, infini et disjoint des autres ensembles de base déjà introduits.

5° $\mathbf{V}$ est une ensemble de symboles de variables.

Les annotations de variables sont définies comme suit.

Définition 4.3.10 *Annotations de variables.* L'ensemble $\mathcal{V}$ des annotations de variables est défini de la façon suivante :

$$\mathcal{V} = \{[V_1 : V_2] | (V_1, V_2) \in (\mathbf{V} \cup \{\_\})^2\} \cup \{[V_1 : \& V_2] | (V_1, V_2) \in (\mathbf{V} \cup \{\_\})^2\}$$

□

Définition des filtres. Les filtres sont des motifs auxquels on ajoute des annotations de variables.

Définition 4.3.11 *Filtres d'entrée.* L'ensemble $\mathcal{F}_I$ des filtres d'entrée est défini de la façon suivante :

- si $l \in \mathcal{L}$, et $v \in \mathcal{V}$, alors $l.v \in \mathcal{F}_I$.
- si $\boldsymbol{p} \in \mathbf{N}$, et $v \in \mathcal{V}$, alors $\boldsymbol{p}.v \in \mathcal{F}_I$ et $\&\boldsymbol{p}.v \in \mathcal{F}_I$.
- si $n \in \mathbb{N}^+, l \in \mathcal{L}, (t_1, \ldots, t_n) \in \mathcal{F}_I^n, (\epsilon_1, \ldots, \epsilon_n) \in \mathbf{E}^n$, et $(v, V_1, \ldots, V_n) \in (\mathcal{V})^{n+1}$, alors $l.v\langle \stackrel{\epsilon_1.V_1}{\to} t_1, \ldots, \stackrel{\epsilon_n.V_n}{\to} t_n\rangle \in \mathcal{F}_I$, et $l.v\{\stackrel{\epsilon_1.V_1}{\to} t_1, \ldots, \stackrel{\epsilon_n.V_n}{\to} t_n\} \in \mathcal{F}_I$.

□

Nous utiliserons la notation $f = t[V_1, \ldots, V_n]$ pour parler d'un filtre f, de motif sous-jacent t et de variables $V_1, \ldots, V_n$.

Exemples et raccourcis syntaxiques. Dans toute la suite, nous ferons un usage intensif des filtres et de la notion de filtrage, utilisés notamment en YATL (voir chapitre 5). Les exemples de programmes d'intégration utiliseront la syntaxe « formelle », la syntaxe graphique montrant rapidement des limites. L'écriture complète de filtres étant relativement lourde à gérer, nous introduisons quelques raccourcis syntaxiques qui seront particulièrement utiles pour une meilleure lisibilité.

Le filtre ci-dessous correspond à la syntaxe complète du filtre donné sur la figure 4.10.

$$\begin{array}{l} filtre_1 = \\ set\{\stackrel{\star}{\rightarrow} tuple\{ \rightarrow nom \rightarrow \mathsf{String}.[\_ : N] \\ \qquad\qquad\qquad \rightarrow publi \rightarrow set \stackrel{\star}{\rightarrow} \&\boldsymbol{Publication}.[\_ : \&Pub]\}\} \end{array}$$

Plusieurs conventions vont nous permettre de simplifier l'écriture :

1° les variables apparaîtrons toujours en majuscule (p.ex. N) et les labels en minuscule (p.ex. *nom*) ;

2° lorsqu'il n'y a qu'une seule variable, et que son type est le plus général possible, c'est-à-dire *Any* pour les labels et $\boldsymbol{Yat}$ pour les motifs, il est possible de remplacer :

- l'annotation $\mathsf{Any}.[\_ : V]$ par V ;
- l'annotation $\&\boldsymbol{Yat}.[\_ : \&V]$ par $\&\boldsymbol{V}$;
- l'annotation $\boldsymbol{Yat}.[V : \_]$ par $\boldsymbol{V}$: ;
- et l'annotation $\stackrel{(i,j).[\_:I]}{\rightarrow}$ par $\stackrel{I}{\rightarrow}$;

3° enfin, lorsqu'il y a une ambiguïté, nous utiliserons toujours la syntaxe complète.

Par exemple, le motif :

$$\begin{array}{l} filtre_2 = \\ set\{\stackrel{\star}{\rightarrow} tuple\{ \rightarrow nom \rightarrow N \\ \qquad\qquad\qquad \rightarrow publi \rightarrow set \stackrel{\star}{\rightarrow} \&\boldsymbol{Pub}\}\} \end{array}$$

est équivalent au motif :

$$\begin{array}{l} filtre'_2 = \\ set\{\stackrel{\star}{\rightarrow} tuple\{ \rightarrow nom \rightarrow \mathsf{Any}.[\_ : N] \\ \qquad\qquad\qquad \rightarrow publi \rightarrow set \stackrel{\star}{\rightarrow} \&\boldsymbol{Yat}.[\_ : \&Pub]\}\} \end{array}$$

4.3.4 Note bibliographique et résultats de complexité

Une formalisation alternative, légèrement plus générale, du modèle en tant que système de types pour les données semistructurées a été récemment proposée dans [BM99]. Cette formalisation se fonde sur l'utilisation d'expressions régulières pour décrire la relation entre les nœuds et leurs fils. Elle a permis d'obtenir les résultats important ci-dessous.

Résultats de complexité pour l'algorithme de typage. [BM99] fournit la complexité d'une version simplifiée de l'instanciation, lorsque les motifs instances sont concrets. On peut démontrer que vérifier le typage d'une forêt concrète par une autre forêt est NP-complet, dans le cas général. Par ailleurs, la complexité devient polynomiale dès lors que les prédicats logiques utilisés pour les annotations sont *discriminants*. C'est à dire lorsque tout couple de prédicats sur des nœuds frères (dans une collection non ordonnée) ou tout couple de prédicats sur des nœuds successifs (dans une collection ordonnée) ont des domaines disjoints. Ce résultat est particulièrement intéressant car vérifié dans la plupart des cas pratiques, notamment lorsque le type est issu d'une source (p.ex. l'union marquée, les n-uplets, etc, vérifient cette propriété). Il démontre la possibilité de fournir une évaluation efficace du filtrage.

Fermeture du système de type. Un autre résultat important concerne la fermeture du système de type par rapport aux opérations ensemblistes élémentaires : union, intersection et complémentation. Dans le cas de la complémentation par exemple, si F est une forêt YAT de domaine $Dom_C(F)$, alors il existe toujours une forêt F' de domaine $Dom_C(F') = \mathcal{F} \setminus Dom_C(F)$. Nous verrons que cette propriété est importante pour assurer l'existance de certaines formes de filtrage (cf. section 5.4).

Cette formalisation demeure, elle aussi, préliminaire. En particulier, elle ne permet pas de capturer toute la généralité du mécanisme d'instanciation. Elle ouvre cependant des opportunités intéressantes dans la perspective d'une étude plus poussée des propriétés théoriques du modèle.

Chapitre 5

Intégration de données

'Die Grenzen meiner Sprache'
bedeuten die Grenzen meiner Welt.
– Ludwig WITTGENSTEIN
(in *Tractatus logico-philosophicus*, 1921)

LE MODÈLE DE DONNÉES INTRODUIT dans le chapitre précédent, nous offre le moyen de représenter l'information devant être traitée tout au long du processus d'intégration. Nous avons maintenant besoin d'un langage pour décrire, au niveau du médiateur (cf. figure 2.7), l'intégration proprement dite. Il est bien entendu possible d'utiliser pour cela un langage de programmation quelconque. Cependant, un langage dédié ainsi qu'une approche déclarative offrent un certain nombre d'avantages. Un langage dédié, plus ciblé, permet de simplifier l'écriture du programme d'intégration et de fournir plus de contrôle sur celui-ci. Le caractère déclaratif du langage permet d'obtenir plus d'indépendance vis-à-vis de l'évaluation, libérant le concepteur des détails d'implantation. En particulier, les techniques d'optimisation étudiées dans le chapitre 6 pourront être mises en œuvre de façon transparente.

Dans ce chapitre, nous décrivons YATL, un langage déclaratif pour la spécification de programmes d'intégration. YATL n'est pas un langage de programmation complet mais il est conçu pour décrire des familles de transformations sur les données semistructurées. Il propose un compromis entre expressivité (*Quelles transformations est-il capable de décrire ?*), simplicité (*Est-il difficile d'écrire un programme ? Celui-ci possède-t'il une sémantique intuitive ?*), contrôle (*Quelles vérifications peut-on faire sur le programme ?*), et performance (*Peut-on fournir une évaluation efficace du programme ?*). Les trois premiers points sont illustrés dans ce chapitre, à travers la description des transformations possibles en YATL, sa sémantique intuitive – dont la nature « graphique » sera évoquée – et son typage. Son évaluation complète ainsi que les techniques d'optimisation seront décrites dans le chapitre 6.

5.1 YATL : un langage pour l'intégration

Nous allons être amenés à faire des choix concernant les familles de transformations que nous pourrons faire. Avant de rentrer dans le vif du sujet, nous procédons brièvement au bilan des besoins en matière d'intégration, ce qui suffira à justifier ces choix d'un point de vue applicatif. Une justification plus rigoureuse nécessiterait une étude des propriétés théoriques du langage (pouvoir d'expression, complexité) qui sort du cadre de cette thèse.

5.1.1 Cahier des charges

Le langage d'intégration doit décrire des restructurations de données dans le modèle YAT. Étant donné un graphe semistructuré, il doit permettre d'extraire les informations utiles, de mettre en relation des informations dispersées dans ce graphe et enfin, de construire un nouveau graphe comme résultat. Les fonctionnalités qu'il doit fournir sont donc essentiellement celles d'un langage de requête et il semble naturel de s'en inspirer. Plus précisément, le langage d'intégration doit être capable de :

1. naviguer à partir d'un point d'entrée dans le graphe ;
2. éliminer les informations inutiles en fonction de plusieurs critères (valeur des données, structure, etc) ;
3. conserver les valeurs obtenues afin de pouvoir les réutiliser ;
4. mettre en relation des informations extraites à plusieurs endroits ;
5. construire un nouveau graphe comme résultat ;
6. calculer de nouvelles informations si besoin est ;
7. offrir des primitives pour la manipulation des collections, en particulier pour traiter l'ordre et effectuer des regroupements ;
8. offrir une forme de récursion ;
9. conserver la correspondance entre données d'entrée et données de sortie.

Les cinq premières fonctionnalités correspondent respectivement à la projection (ou à la navigation), à la sélection (clause `where`), à la liaison de variables et à la jointure (clause `from`), et enfin à la création du résultat (clause `select`) que l'on trouve dans tout langage de requête digne de ce nom. La possibilité de rajouter des informations à l'aide de fonctions quelconques (point n° 6) est nécessaire, en conjonction avec la jointure, pour mettre en relation des informations représentées de façon différente et traiter ainsi les problèmes d'hétérogénéité. Fournir des primitives évoluées pour le traitement des collections ordonnées (point n° 7) est indispensable pour exploiter les formats documentaires. Une forme de récursion (point n° 8) permet de travailler sur des structures arbitrairement imbriquées ce qui, nous le verrons, est nécessaire à l'écriture de transformations génériques. Enfin, nous verrons que la sauvegarde des correspondances (point n° 9) est une propriété importante pour optimiser les accès par navigation dans la vue intégrées. Cela peut également faciliter la propagation des mises à jour [CH95] ; indiquons cependant que les problèmes liés aux mises à jour ne seront pas traités ici.

Le reste du chapitre est organisé comme suit. En guise d'entrée en matière, la section 5.1.2 propose un premier exemple de règle YAT_L. À l'aide d'une représentation graphique, la section 5.1.3 propose pour cette règle une sémantique intuitive. La section 5.1.4 illustre ensuite son évaluation. Les fonctionnalités plus avancées relatives à la manipulation des collections, à la création d'identifiants, au traitement de l'hétérogénéité sémantique et à la description de programmes génériques sont présentées respectivement dans les sections 5.1.5, 5.1.6, 5.1.7 et 5.1.8.

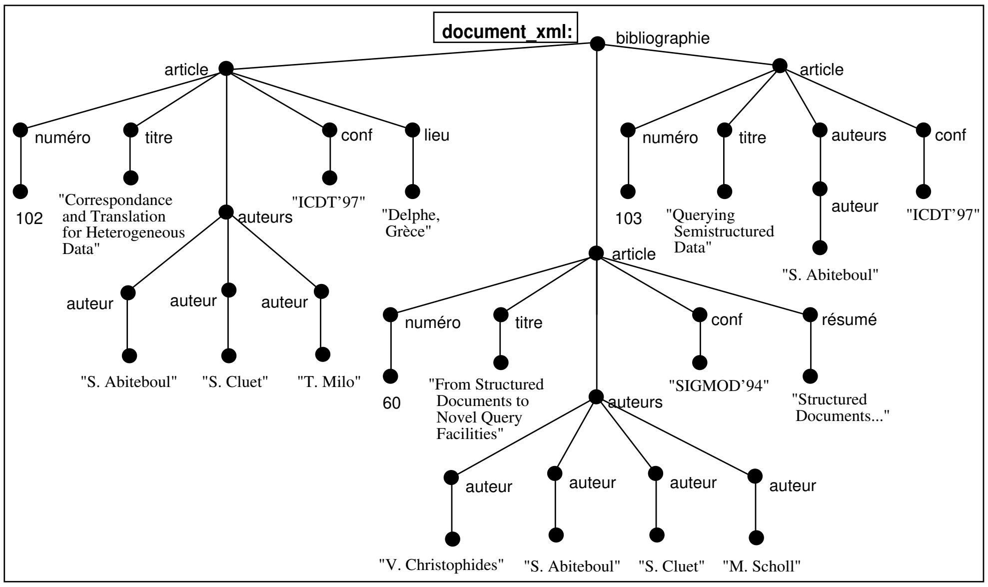


FIG. 5.1: *Un nouveau document XML avec des champs additionnels*

5.1.2 Filtrage et restructuration en YATL : un premier exemple

Un programme YATL consiste en un ensemble de règles décrivant chacune une partie de l'intégration. Chaque règle est constituée d'un *corps*, comprenant un ensemble de *filtres d'entrée*, de *prédicats booléens* et de *fonctions externes*, et d'une *tête* comprenant un *filtre de sortie*.

Pour les besoins de la présentation, nous imaginons qu'il nous faut créer une base de données à objets contenant des informations sur les chercheurs en informatique et sur leurs publications. Les informations sont extraites d'un document bibliographique XML décrit sur la figure 5.1 (correspondant à la structure XML proposée dans la figure 4.6) et de plusieurs bases de données relationnelles. Celles-ci contiennent des informations relatives aux personnels de différents centres de recherche ou stockent les actes de conférences. La règle article_vers_publication ci-dessous donne un avant-goût de YATL : elle crée un objet publication pour chaque article du document XML.

rule article_vers_publication =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \xrightarrow{\star} \boldsymbol{A}: \\
 & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\boldsymbol{Publication}(\&\boldsymbol{A}): & & \qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\textit{class} \rightarrow \textit{publication} \rightarrow \textit{tuple} & & \qquad \rightarrow \textit{auteurs} \xrightarrow{\star} \textit{auteur} \rightarrow Au, \\
\quad \{ \rightarrow \textit{titre} \rightarrow Ti, & \Longleftarrow & \qquad \rightarrow \textit{conf} \rightarrow Co, \\
\quad\ \rightarrow \textit{année} \rightarrow An, & & \qquad \xrightarrow{\star} \boldsymbol{Champ}\rangle, \\
\quad\ \rightarrow \textit{auteurs} \rightarrow \textit{list} \xrightarrow{\star} Au\} & & An = \mathsf{extraire\_année}(Co), \\
 & & An > 1995
\end{array}
$$

Le corps de la règle est situé à droite du symbole $\Longleftarrow$ et contient un filtre d'entrée et deux fonctions externes. Le rôle d'un filtre d'entrée est d'extraire les données désirées à partir d'un point du graphe. Dans l'exemple, il est utilisé pour récupérer les valeurs utiles (numéro de l'article, titre, auteurs, nom de la conférence, etc), à partir de la racine de persistance `document_xml`, dans des variables (respectivement Num, Ti, Au, Co, et $\boldsymbol{A}$). Rappelons que, comme nous l'avons vu dans la section 4.2, le filtrage en YATL est fondé sur la notion d'instanciation et capture trois fonctionnalités essentielles : la navigation dans le graphe, la sélection par rapport au type des données et la liaison des variables (voir aussi les trois premiers points de notre cahier des charges). Notons également que la variable $\boldsymbol{A}$ sera liée aux sous-motifs contenant les articles, et que $\boldsymbol{\&A}$ correspond donc à l'identifiant de $\boldsymbol{A}$. Lorsque plusieurs filtres d'entrée sont utilisés dans la règle, on utilise un produit cartésien sur les liaisons que chacun d'eux retourne. Cela permet notamment d'exprimer la jointure : nous aurons besoin de cette fonctionnalité pour la résolution de l'hétérogénéité (cf section 5.1.7).

Les fonctions externes sont utilisées pour calculer de nouvelles valeurs lorsque ce calcul sort du cadre du langage d'intégration proprement dit. Dans notre exemple, la fonction externe (extraire_année) calcule l'année de publication à partir du descriptif de la conférence et met le résultat dans la variable An. Il s'agira toujours de fonctions sans effets de bord. Enfin, l'utilisation d'un prédicat booléen (An > 1995) sert à sélectionner uniquement les articles postérieurs à 1995.

Les filtres de sortie ont pour rôle de décrire la partie du graphe que l'on souhaite créer, à partir des variables utilisées dans le corps de la règle. YATL offre des filtres de sortie très puissants, pouvant inclure des primitives de réordonnancement et de regroupement, ainsi que des fonctions d'agrégation. De plus, chaque filtre de sortie peut être associé à une fonction de Skolem [1] permettant de nommer les motifs créés. Dans notre exemple, le filtre crée un nouvel objet pour chaque identifiant d'article (grâce à la primitive $\boldsymbol{Publication}(\boldsymbol{\&A})$). Les valeur des motifs ont, quant à elles, la structure d'un objet ODMG de la classe Publication, avec un attribut titre (tiré de la variable Ti), une année de publication (tirée de la variable An) et une liste d'auteurs (dans la variable Au).

5.1.3 Description graphique des transformations : une sémantique intuitive

Avant d'explorer les fonctionnalités avancées du langage, nous essayons auparavant de donner au lecteur une idée plus claire de sa sémantique. Pour ce faire, nous procédons en deux étapes : nous expliquons dans cette section l'intuition qui se cache derrière la règle YATL de la section précédente puis, dans la section suivante, nous procédons, pas à pas, à son évaluation.

La conception de YATL a été fortement influencée par [ACM97] et l'intuition graphique présente dans la notion de correspondance. Cette influence se traduit dans le langage par le rôle fondamental joué par les filtres et la nature très visuelle de ceux-ci. L'idée originale est de décrire la transformation désirée en mettant en relation les structures d'entrée et de sortie puis en inférant un programme qui effectue la transformation qui correspond à cette relation. La figure 5.2 illustre une telle « mise en correspondance » entre notre article XML (à droite) et la classe ***Publication*** (à gauche).

Notons, tout d'abord, la présence (en bas à droite de la figure) de la fonction externe extraire_année ainsi que du prédicat de sélection, qui sont sans aucun doute les éléments du

1. Bien que YATL s'éloigne d'une sémantique logique classique, la similitude avec des formules en forme de Skolem justifie cet abus de langage (que l'on retrouve par ailleurs dans [FFK+98]). Voir également la note au bas de la page 38...

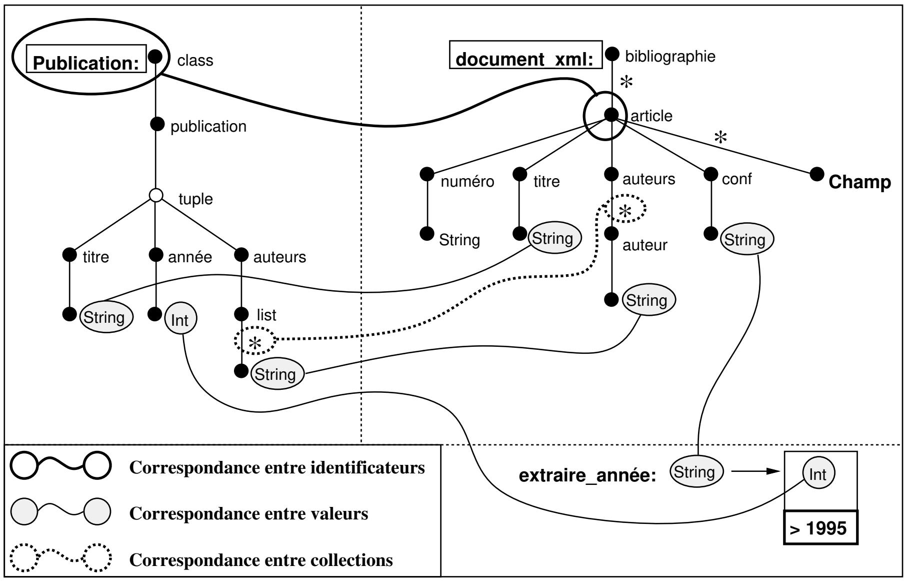


FIG. 5.2: *Une description graphique à l'aide de correspondances*

langage les moins naturels à représenter graphiquement. Ensuite, on remarque l'utilisation de plusieurs types de correspondances. La première (notée) met en relation les identificateurs des arbres, ce qui permet de déterminer quand la création de l'un doit avoir lieu en fonction de l'autre. Dans notre exemple, il s'agit d'une relation entre les nœuds racines (ou identifiants des motifs) de part et d'autre, mais il pourra s'agir d'une relation plus complexe, par exemple entre une valeur (*clé*) et un nœud. Dans YATL, ceci se traduit par l'utilisation d'une fonction de Skolem (***Publication***(**&*A***)) qui paramètre la création d'identifiants.

Le deuxième type de correspondance (noté) met en relation deux labels et indique que leurs valeurs respectives doivent être égales (p.ex. le titre de l'article XML et celui de l'objet publication). Notons que celle-ci ne peut être valide que si les types respectifs des labels sont compatibles. Dans certains cas, cette correspondance est réalisée par l'intermédiaire d'une fonction externe (p.ex. allant du nom de la conférence dans l'article vers l'année de publication). En YATL, une telle correspondance se traduit par l'usage de la même variable entre un filtre d'entrée et de sortie (p.ex. Ti pour le titre) ou entre un filtre et une fonction externe (p.ex. Co et An pour la fonction extraire_année).

Enfin, le troisième type de correspondance (noté) indique une relation entre deux collections. Dans l'exemple, elle ne paraît pas indispensable dans la mesure où la relation entre les noms d'auteurs d'une part, et le type des filtres d'autre part, suffisent pour comprendre la sémantique de la transformation. Mais il n'en va pas toujours de même, lorsque plusieurs collections sont impliquées. Par ailleurs, il est important de vérifier la cohérence entre les correspondances de labels et correspondances de collections pour éviter les conflits. Par exemple, le haut de la figure 5.3 montre une mise en correspondance incohérente. Les noms des auteurs d'articles en XML sont mis en relation avec l'attribut titre de l'objet, mais celui-ci ne peut hélas pas contenir une collection : dans le cas où il y a plusieurs auteurs, la sémantique de la transformation n'est pas claire ! Si l'on choisit malgré tout d'autoriser ce genre de correspondance, cela nous mène tout droit vers une sémantique indéterministe.

Au bas de la figure 5.3, le problème est différent. Il faut d'abord observer qu'il doit

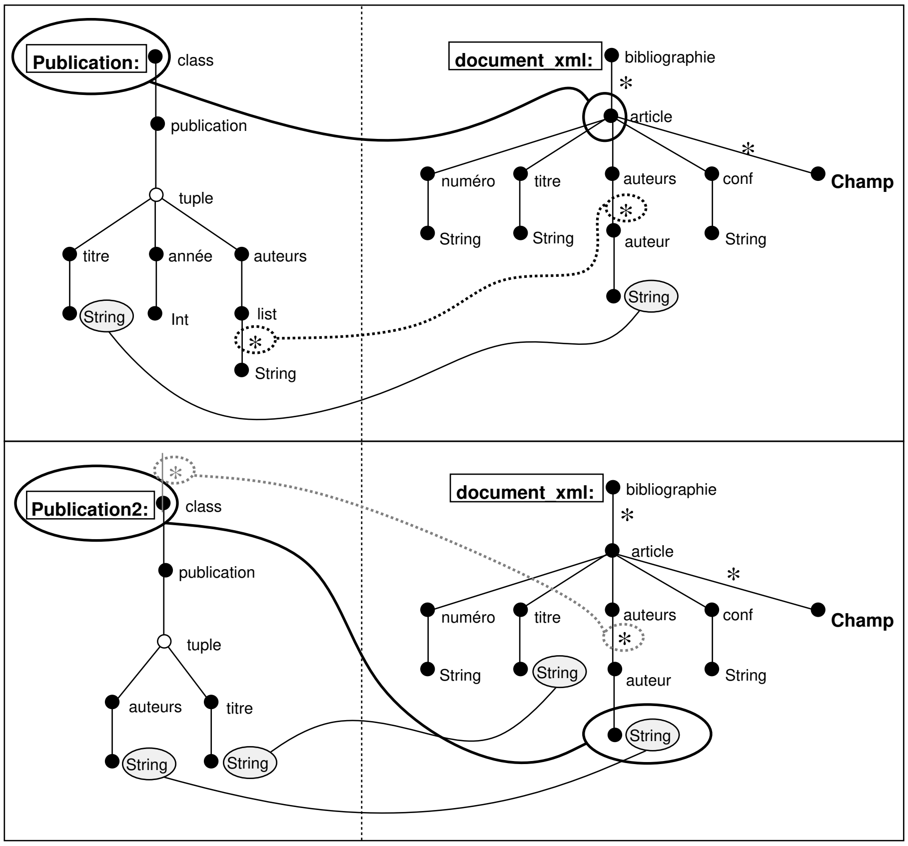


FIG. 5.3: *Les limites d'une description graphique...*

y avoir un nouveau motif pour chaque auteur distinct de l'article (correspondance entre identificateurs). Il y a donc implicitement une correspondance entre la collection d'auteurs et l'ensemble des motifs créés (indiquée en pointillés sur la figure). Enfin, la relation entre les labels titre implique que le même titre se retrouve sur tous les arbres créés pour une publication donnée. Il n'y a pas nécessairement d'incohérence. Malheureusement, si on applique cette transformation à plusieurs articles, comme c'est le cas ici, on ne sait plus quelle doit être la valeur du titre pour un auteur fixé.

On voit clairement que les différentes sémantiques possibles (regroupement, jointures, imbrication) lors du traitement des collections peuvent induire des ambiguïtés ou des incompatibilités. L'utilisation d'une spécification graphique montre ici ses limites. Nous reviendrons sur ces problèmes d'un point de vue du langage lors de l'étude des fonctionnalités liées à la manipulation des collections.

Quelques remarques pour conclure cette rapide présentation. Tout d'abord, la description par correspondance s'attache à conserver autant que possible les symétries qui existent entre motifs d'entrée et de sortie. Lorsqu'une symétrie complète peut être trouvée, il est alors possible de déduire les programmes de conversion dans les deux sens [ACM97] à partir des correspondances, voire d'inférer les correspondances elles-mêmes [Zoh97, MZ98b]. Dans l'exemple de la figure 5.2, la règle YAT$_L$ article_vers_publication de la section précédente exprime la transformation de l'article XML vers la publication objet. Par contre, deux facteurs nous empêchent d'écrire la transformation inverse : d'une part, certaines informations des articles XML n'ont pas de correspondant (p.ex. le numéro de l'article), d'autre part, la fonction externe ne peut opérer que dans un seul sens (il est *a priori* impossible de retrouver le nom de la conférence à partir de l'année où elle a eu lieu). La spécification graphique de règles YATL a été étudiée par Katarzyna SMAGA [Sma97]. Une interface graphique implantée en Java fournit des fonctionnalités d'édition et de mise en correspondance de motifs dans les règles YATL.

5.1.4 Et maintenant, évaluons !

Observons maintenant comment se passe l'évaluation. Nous partons de l'ensemble d'articles XML de la figure 5.1. La règle article_vers_publication est exécutée en appliquant les étapes suivantes :

1° Le motif concret d'entrée (ici l'ensemble d'articles XML) est apparié avec les filtres du corps de la règle. Il s'agit d'une opération de filtrage simple, avec la sémantique de produit cartésien sur les variables présentées dans la section 4.2. Elle retourne l'ensemble de liaisons suivant [2] (dans lequel les $\boldsymbol{a_i}$ sont les sous-motifs des articles) :

{ [$\boldsymbol{A} = \boldsymbol{a1}$; $Num = 103$; Ti = "Querying..."; Au = "S. Abiteboul";
Co = "ICDT'97"],
[$\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; Au = "V. Christophides";
Co = "SIGMOD'94"],
[$\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; Au = "S. Abiteboul";
Co = "SIGMOD'94"],
[$\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; Au = "S. Cluet";
Co = "SIGMOD'94"],
[$\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; Au = "M. Scholl";
Co = "SIGMOD'94"],
[$\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; Au = "S. Abiteboul";
Co = "ICDT'97"],
[$\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; Au = "S. Cluet";
Co = "ICDT'97"],
[$\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; Au = "T. Milo";
Co = "ICDT'97"] }

2° La fonction externe extraire_année est évaluée sur chaque n-uplet rajoutant à chacun d'eux une nouvelle liaison pour la variable An :

{ [$An = 1997$; $\boldsymbol{A} = \boldsymbol{a1}$; $Num = 103$; Ti = "Querying..."; ...],
[$An = 1994$; $\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; ...],
[$An = 1994$; $\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; ...],
[$An = 1994$; $\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; ...],
[$An = 1994$; $\boldsymbol{A} = \boldsymbol{a2}$; $Num = 60$; Ti = "From..."; ...],
[$An = 1997$; $\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; ...],
[$An = 1997$; $\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; ...],
[$An = 1997$; $\boldsymbol{A} = \boldsymbol{a3}$; $Num = 102$; Ti = "Correspondance..."; ...] }

2. Pour des raisons de lisibilité, nous représentons ici les structures de Tab comme des collections de n-uplets.

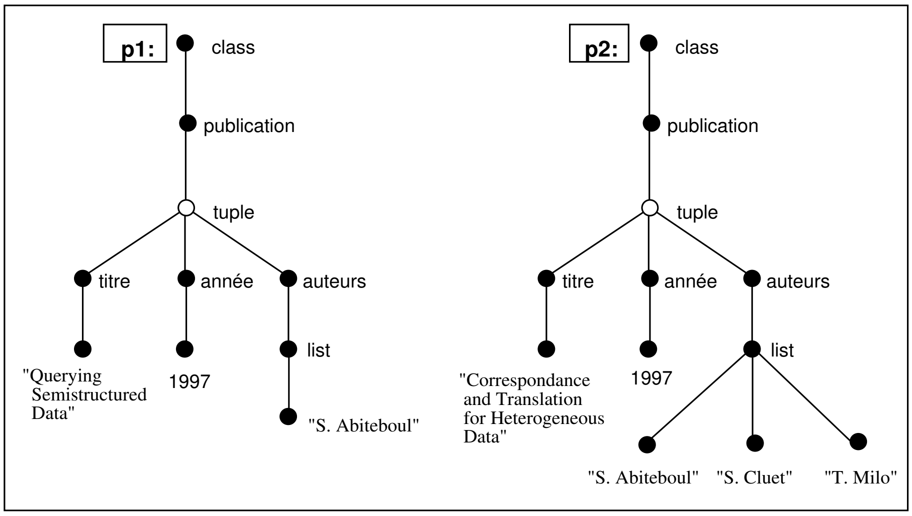


FIG. 5.4: *Résultat de la règle* article_vers_publication

3° Puis le prédicat sur l'année est appliqué, ce qui élimine les informations relatives aux publications antérieures à 1995 :

$$\begin{array}{lllll}
\{ [\ An = 1997; & \boldsymbol{A} = \boldsymbol{a1}; & Num = 103; & Ti = \text{"Querying..."}; & \ldots\ ], \\
\ \ [\ An = 1997; & \boldsymbol{A} = \boldsymbol{a3}; & Num = 102; & Ti = \text{"Correspondance..."}; & \ldots\ ], \\
\ \ [\ An = 1997; & \boldsymbol{A} = \boldsymbol{a3}; & Num = 102; & Ti = \text{"Correspondance..."}; & \ldots\ ], \\
\ \ [\ An = 1997; & \boldsymbol{A} = \boldsymbol{a3}; & Num = 102; & Ti = \text{"Correspondance..."}; & \ldots\ ] \}
\end{array}$$

4° Le filtre de sortie est ensuite instancié sur les liaisons restantes. Notons que cela entraîne à la fois l'évaluation de la fonction de Skolem (ici $\boldsymbol{Publication}(\&\boldsymbol{A})$), qui crée les identifiants, et du motif lui-même, qui calcule la valeur associée à ces identifiants. Le résultat de cette opération comprend deux arbres représentés sur la figure 5.4. La fonction de Skolem est bel et bien appliquée sur les identifiants des nœuds racines des articles, comme on s'y attendait. La valeur de $\&\boldsymbol{A}$ étant effectivement égale aux $\&\boldsymbol{a_i}$.

Notons que ce processus d'évaluation ne prétend pas être le plus efficace possible : on pourrait par exemple éliminer très tôt les informations inutiles (Num) ou éviter le produit cartésien sur les noms d'auteurs. Les différentes alternatives à cette évaluation seront étudiées dans le chapitre 6.

5.1.5 Manipulation des collections

Tout comme les autres langages utilisés pour l'intégration dans un contexte semistructuré (StruQL [FFK+98] et MSL [PAG96]), YAT$_L$ travaille sur des ensembles, cependant, une fonctionnalité originale du langage est d'offrir des primitives pour le traitement des collections ordonnées et avec duplicats.

Préserver et modifier l'ordre. Nous avons vu dans le chapitre précédent, que le modèle de données distinguait entre fils ordonnés ou pas. Sans la possibilité d'exploiter cette propriété dans le langage, cela ne présente cependant qu'un intérêt réduit. Deux constructions du langage permettent de manipuler l'ordre : en entrée, il est possible de récupérer l'information relative à l'ordre existant en la stockant dans une variable ; en sortie, il est possible d'imposer un ordre en fonction de critères qui utilisent les valeurs des variables.

La règle article_vers_publication_ordonné ci-dessous, modifie la règle de la section 5.1.2 dans le but de préserver l'ordre original des auteurs.

<u>rule article_vers_publication_ordonné</u> =

$$\begin{array}{lcl}
 & & \texttt{document_xml} : \\
 & & \textit{bibliographie} \xrightarrow{\star} \boldsymbol{A} : \\
 & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\boldsymbol{Publication}(\&\boldsymbol{A}) : & & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\textit{class} \rightarrow \textit{publication} \rightarrow \textit{tuple} & & \qquad\qquad \rightarrow \textit{auteurs} \xrightarrow{I} \textit{auteur} \rightarrow Au, \\
\qquad \{ \rightarrow \textit{titre} \rightarrow Ti, & \Longleftarrow & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
\qquad\quad \rightarrow \textit{année} \rightarrow An, & & \qquad\qquad \xrightarrow{\star} _\rangle, \\
\qquad\quad \rightarrow \textit{auteurs} \rightarrow \textit{list} \xrightarrow{o(I)} Au\} & & An = \mathsf{extraire_année}(Co), \\
 & & An > 1995
\end{array}$$

Dans cette règle, on observe deux changements :

1° Dans le filtre d'entrée, l'utilisation de la variable I permet de stocker le rang de l'arête (cf. section 4.2.1) et donc de conserver l'information sur l'ordre.

2° Dans le filtre de sortie, on remarque l'utilisation de cette même variable dans le critère d'ordonnancement, $o(I)$, situé sur l'arête qui crée la liste d'auteurs. Ceci permet d'imposer la création des fils dans l'ordre fixé par la valeur de I et ainsi de restituer l'ordre tel qu'il a été stocké.

Cette règle permet donc de conserver, au cours de l'intégration, les informations sur l'ordre lorsque celui-ci possède une valeur sémantique importante. Dans notre exemple, l'ordre dans lequel les auteurs apparaissent peut ainsi être conservé. Notons que lorsque le symbole $\star$ est utilisé, la sémantique de la règle ne spécifie aucun ordre *a priori*. Notamment, l'implantation du langage n'assure en aucun cas la conservation d'un ordre de départ existant. Celui-ci peut être modifié au cours de l'évaluation, en particulier pour des raisons d'efficacité (voir le chapitre sur l'optimisation).

Regardons maintenant comment effectuer des réordonnancements suivant d'autres critères. À partir d'un ensemble d'articles, la règle ci-dessous crée une liste contenant les même informations que les objets publications, mais ordonnées par année, puis par numéro. Le critère d'ordonnancement est ici $o(An, Num)$. Lorsqu'il n'y a qu'une seule variable, c'est bien entendu l'ordre naturel qui est employé (p.ex. alphabétique pour les chaînes de caractères). Dans le cas de plusieurs variables, comme ici, c'est l'ordre lexicographique sur celles-ci qui est utilisé. La valeur générée par la règle toutes_les_publications_par_année est donc une liste classée par années croissantes et pour chaque année par numéros d'articles croissants (même si ce numéro n'est pas conservé par la suite).

$$\underline{\textsf{rule toutes\_les\_publications\_par\_année}} =$$

$$\begin{array}{ll}
 & \texttt{document_xml :} \\
 & \mathit{bibliographie} \xrightarrow{\star} \\
 & \quad \mathit{article}\langle \rightarrow \mathit{numéro} \rightarrow Num, \\
list \overset{o(An,Num)}{\rightarrow} tuple\{ \rightarrow \mathit{titre} \rightarrow Ti, & \qquad\quad \rightarrow \mathit{titre} \rightarrow Ti, \\
\qquad\qquad \rightarrow \mathit{année} \rightarrow An\} \Longleftarrow & \qquad\quad \rightarrow \mathit{auteurs} \xrightarrow{I} \mathit{auteur} \rightarrow Au, \\
 & \qquad\quad \rightarrow \mathit{conf} \rightarrow Co, \\
 & \qquad\quad \xrightarrow{\star} _\rangle, \\
 & An = \textsf{extraire_année}(Co), \\
 & An > 1995
\end{array}$$

Regroupement. Le deuxième type de primitive dont nous avons besoin pour le traitement des collections concerne le regroupement. Il s'agit d'une fonctionnalité d'autant plus importante que l'on manipule des structures de données arborescentes donc naturellement imbriquées.

Par exemple, si l'on souhaite obtenir la liste des publications par auteurs, il est nécessaire d'aplatir l'ensemble des informations contenues dans les articles d'entrée (ce qui est réalisé par la sémantique de produit cartésien sur les variables du filtre), puis de regrouper en fonction du nom de l'auteur. Un telle transformation est décrite par la règle ci-dessous.

$$\underline{\textsf{rule toutes\_les\_publications\_par\_auteurs}} =$$

$$\begin{array}{ll}
 & \texttt{document_xml :} \\
 & \mathit{bibliographie} \xrightarrow{\star} \\
 & \quad \mathit{article}\langle \rightarrow \mathit{numéro} \rightarrow Num, \\
set\{ \overset{g(Au)}{\rightarrow} tuple\{ \rightarrow \mathit{auteur} \rightarrow Au, & \qquad\quad \rightarrow \mathit{titre} \rightarrow Ti, \\
\qquad\qquad \rightarrow \mathit{publis} \rightarrow list \overset{o(Num)}{\rightarrow} Num\}\} \Longleftarrow & \qquad\quad \rightarrow \mathit{auteurs} \xrightarrow{\star} \mathit{auteur} \rightarrow Au, \\
 & \qquad\quad \rightarrow \mathit{conf} \rightarrow Co, \\
 & \qquad\quad \xrightarrow{\star} _\rangle, \\
 & An = \textsf{extraire_année}(Co), \\
 & An > 1995
\end{array}$$

La phase d'évaluation du corps de la règle est identique à celle indiquée dans la section 5.1.4 et retourne la même structure de Tab qu'à la fin de la troisième étape, page 74. Par contre, l'annotation $g(Au)$ dans le filtre de sortie impose un regroupement sur la variable Au. Ainsi, le motif généré (montré sur la figure 5.5) présente un fils pour chaque auteur, et chacun de ces fils est construit uniquement avec les informations relatives à l'auteur en question. Pour chacun d'eux, on construit un n-uplet avec son nom et la liste (ordonnée) des numéros de ses publications. Dans le cas où aucun critère de regroupement n'est fourni explicitement (lorsque une arête porte l'annotation $\star$), on n'effectue aucun regroupement, c'est-à-dire que l'on crée un fils pour chaque liaison dans le tab.

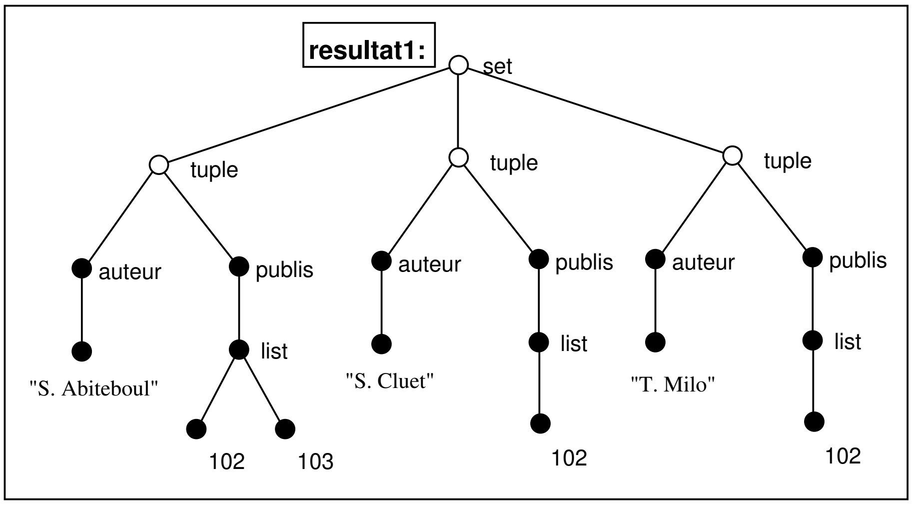


FIG. 5.5: *Regroupement sur les auteurs*

Regroupement et non déterminisme. Cela semble être le bon moment pour se souvenir des remarques de la section 5.1.4 concernant les incohérences possibles lors de la manipulation de collections. Si l'on regarde une fois de plus la règle toutes_les_publications_par_auteurs, on s'aperçoit que l'utilisation du nom de l'auteur dans le n-uplet n'est possible qu'en raison de l'utilisation de la primitive de regroupement $g(Au)$ qui assure qu'il n'y aura bien qu'une seule valeur pour l'auteur. Remplacer la variable Au par la variable Ti comme c'est le cas sur la règle ci-après, n'aurait aucun sens car il y a bien entendu plusieurs titres par auteurs.

<u>rule toutes_les_publications_par_auteurs_erronée</u> =

$$
\begin{array}{ll}
 & \texttt{document\_xml :} \\
 & \textit{bibliographie} \xrightarrow{\star} \\
 & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\textit{set}\{\overset{g(Au)}{\rightarrow} \textit{tuple}\{ \rightarrow \textit{titre} \rightarrow Ti, & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\qquad\qquad \rightarrow \textit{publis} \rightarrow \textit{list} \overset{o(Num)}{\rightarrow} Num\}\} \Longleftarrow & \qquad\qquad \rightarrow \textit{auteurs} \xrightarrow{\star} \textit{auteur} \rightarrow Au, \\
 & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
 & \qquad\qquad \xrightarrow{\star} \_\rangle, \\
 & An = \text{extraire\_année}(Co), \\
 & An > 1995
\end{array}
$$

Pour résoudre ce problème, plusieurs solutions peuvent être adoptées. La première consiste à choisir une sémantique indéterministe : lorsqu'il y a plusieurs valeurs possibles, l'une d'elles est sélectionnée au hasard par le système. L'autre politique consiste à interdire ce genre de règle, ce qui implique de pouvoir détecter les cas où le déterminisme ne peut être assuré. La contrainte suivante est une condition nécessaire et suffisante du déterminisme d'une règle :

Une variable V peut annoter le nœud d'un filtre de sortie, si et seulement si, V dépend fonctionnellement de l'ensemble des variables impliquées dans les regroupements situés sur les arêtes entre la racine du filtre et le nœud de V.

La notion de dépendance fonctionnelle est celle couramment utilisée dans les bases de données : une variable V' dépend d'un ensemble de variables $\vec{V}$ (noté $\vec{V} \triangleright V'$), si et seulement si, quelque soit $(\vec{v_1}, v'_1)$ et $(\vec{v_2}, v'_2)$ deux assignations des variables $(\vec{V}, V')$, alors $\vec{v_1} = \vec{v_2}$ implique $v'_1 = v'_2$. Notons que l'annotation $\star$ s'apparente à un regroupement sur toutes les variables. En particulier, on peut toujours utiliser une variable après une telle annotation.

Le problème est donc désormais de déterminer les dépendances fonctionnelles d'une variable. Il y a deux façon d'obtenir cette information :

- soit lorsqu'une connaissance sémantique supplémentaire est disponible (p.ex. clés relationnelles, fournie par l'utilisateur) ;
- soit lorsqu'elle découle du filtrage, quand l'une des variables est un identificateur.

Par exemple, dans la règle précédente, $\{\&\boldsymbol{A}\} \triangleright Ti$, car les arêtes qui relient le nœud racine de l'article et la variable titre sont simples. Il ne peut donc pas y avoir plusieurs titres pour un article donné. Autre exemple, si le numéro d'un article est une clé, alors $\{Num\} \triangleright \&\boldsymbol{A}$.

Manipulation de vecteurs et agrégation. La capacité à traiter les collections ordonnées est importante dans de nombreux domaines d'application. Jusqu'ici, tous les exemples que nous avons choisis reflètent des cas de figure usuels en bases de données. Les primitives de YATL permettent également de manipuler des vecteurs. À titre d'exemple, la règle ci-dessous calcule la transposée d'une matrice $n \times m$ quelconque ! On peut remarquer que contrairement aux autres règles que nous avions écrites, celle-ci ne part pas d'une racine de persistance. Au contraire, le motif d'entrée $\boldsymbol{Id}$ est passé en paramètres à la règle. Cela nous permettra d'utiliser cette règle sur n'importe quel motif d'entrée. Nous reviendrons sur ce mécanisme dans la section suivante. Pour le moment, intéressons-nous à l'évaluation de cette règle.

<u>rule transposée</u>($\boldsymbol{Id}$) =

$$Mat \overset{g(J)o(J)}{\rightarrow} Y \overset{g(I)o(I)}{\rightarrow} X \rightarrow \boldsymbol{A} \quad \Longleftarrow \quad \begin{array}{c}\boldsymbol{Id}: \\ Mat \overset{I}{\rightarrow} X \overset{J}{\rightarrow} Y \rightarrow \boldsymbol{A}\end{array}$$

Comme on peut le voir sur la figure 5.6, les matrices sont représentées comme des arbres réguliers. La règle récupère tout d'abord l'ensemble des informations à deux niveaux de profondeur. Cela nous donne une structure de Tab avec les liaisons pour les variables I, J, X, Y et $\boldsymbol{A}$ (qui contient la valeur du sous-motif). On procède ensuite à un regroupement (doublé d'un ordonnancement) sur les variables I et J qui intervertit les indices de la matrice. Le résultat est donné à droite de la figure 5.6.

Dans le contexte du traitement de vecteurs, il serait bien entendu utile de pouvoir effectuer des opérations de regroupement. Nous nous contenterons d'indiquer que ce type d'opération peut être trivialement rajouté à YATL. D'un point de vue pratique, de telles opérations sont courantes dans les tableurs ou lors de la génération de rapports [Cha98]. Enfin, notons au passage que l'intérêt pour la manipulation de telles structures va croissant [FM95, LMW96, MS97].

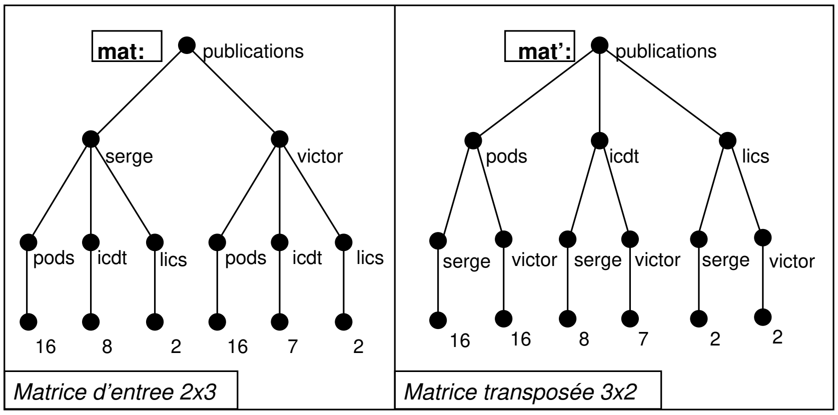


FIG. 5.6: *Transposition d'une matrice* 2×3

5.1.6 Création d'identifiants et sémantique globale

Jusqu'à présent, nous n'avons présenté que des programmes YATL ne contenant qu'une seule règle. Nous nous intéressons maintenant à la sémantique d'un ensemble de règle et aux mécanismes permettant de faire le lien entre chacune d'elles. Pour cela, YATL utilise des fonctions de Skolem explicites dans les filtres de sortie et offre la possibilité de définir des règles paramétrées pouvant être appelées à partir d'autres règles.

Manipulation de références et partage d'information. Les fonctions de Skolem sont des entités globales dans un programme YATL, c'est à dire qu'elles sont connues dans toutes les règles. Elles permettent de contrôler la création de nouveaux motifs et de créer des références entre eux. Il est ainsi possible de réutiliser les motifs créés d'une règle à l'autre et de partager des informations. Les deux règles suivantes illustrent ce point en créant des objets dans deux classes (***Publication*** et ***Chercheur***) avec des références croisées.

La première règle est identique à celle de la section 5.1.2, à ceci près que les auteurs ne sont plus stockés directement dans un attribut de la classe ***Publication***, mais dans des objets séparés, appartenant à la classe ***Chercheur***. La référence est construite grâce à l'appel de la fonction de Skolem correspondant à l'objet adéquat (ici **&*Chercheur***(Au)).

rule article_vers_publication2 =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \xrightarrow{\star} \boldsymbol{A}: \\
 & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\boldsymbol{Publication}(\&\boldsymbol{A}): & & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\textit{class} \rightarrow \textit{publication} \rightarrow \textit{tuple} & & \qquad\qquad \rightarrow \textit{auteurs} \xrightarrow{I} \textit{auteur} \rightarrow Au, \\
\quad \{ \rightarrow \textit{titre} \rightarrow Ti, & \Longleftarrow & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
\quad\; \rightarrow \textit{année} \rightarrow An, & & \qquad\qquad \xrightarrow{\star} \_\rangle, \\
\quad\; \rightarrow \textit{auteurs} \rightarrow \textit{list} \xrightarrow{o(I)} \&\boldsymbol{Chercheur}(Au)\} & & An = \mathsf{extraire\_année}(Co), \\
 & & An > 1995
\end{array}
$$

Il est possible de paramétrer une fonction de Skolem par plusieurs variables, pouvant contenir des valeurs ou des identifiants. Notons que la sémantique d'une fonction de Skolem est implicitement celle d'un regroupement par rapport à ses paramètres. C'est-à-dire que pour chaque ensemble distinct de valeurs : (i) un nouveau motif est créé, avec son identificateur, (ii) la valeur de ce motif est construite en utilisant les liaisons pour lesquelles la valeur des variables correspond. Dans la règle suivante, cette propriété de regroupement est utilisée lors de la création des objets $\boldsymbol{Chercheur}(Au)$. À partir de l'ensemble d'articles XML, la règle article_vers_chercheur construit un nouveau motif pour chaque nom d'auteur distinct. Grâce à la sémantique de regroupement, chaque nouveau motif ne contient que les informations sur le chercheur en train d'être créé. On peut remarquer que les attributs position et intérêt ne peuvent être complètes car les documents XML ne contiennent pas l'information nécessaire. On leur donne donc une valeur par défaut. Pour obtenir les informations correspondantes, nous allons devoir interroger une autre source (voir section 5.1.7).

$$
\begin{array}{l}
\underline{\text{rule article\_vers\_chercheur}} = \\
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \xrightarrow{\star} \boldsymbol{A}: \\
 & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\boldsymbol{Chercheur}(Au): & & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\textit{class} \rightarrow \textit{chercheur} \rightarrow \textit{tuple} & & \qquad\qquad \rightarrow \textit{auteurs} \xrightarrow{\star} \textit{auteur} \rightarrow Au, \\
\quad \{ \rightarrow \textit{nom} \rightarrow Au, & \Longleftarrow & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
\qquad \rightarrow \textit{position} \rightarrow \text{“”}, & & \qquad\qquad \xrightarrow{\star} \_\rangle, \\
\qquad \rightarrow \textit{intérêt} \rightarrow \textit{list} & & An = \text{extraire\_année}(Co), \\
\qquad \rightarrow \textit{articles} \rightarrow \textit{set}\{\xrightarrow{\star} \boldsymbol{\&Publication}(\boldsymbol{\&A})\}\} & & An > 1995
\end{array}
\end{array}
$$

Notons que dans cette seconde règle, il est indispensable de filtrer les articles postérieurs à 1995 pour s'assurer de la cohérence entre les objets référencés avec ceux créés dans la règle précédente. L'utilisation des fonctions de Skolem dans les deux règles crée des références croisées. L'évaluation de ces deux règles est possible uniquement car la création d'un nouvel identifiant de motif et de sa valeur peuvent se faire indépendamment l'un de l'autre. Finalement, il est intéressant de remarquer que ces deux règles permettent de tirer parti du modèle orienté-objet en ne créant qu'un objet par auteur, contrairement à la règle de la section précédente dans laquelle les informations sur les auteurs étaient dupliqués. Regardons d'abord comment un ensemble de règles peut servir pour peupler une vue.

Racines de persistance en sortie. Jusqu'à présent, toutes les règles que nous avions écrites retournaient un motif YAT, pouvant lui-même contenir un ensemble de motifs nommés à l'aide d'une fonction de Skolem. Lorsque l'on s'intéresse à la sémantique global d'un programme, il faut également prêter attention à fournir des points d'entrée dans la vue intégrée, ainsi que de peupler ces points d'entrée. Pour cela, il est possible de faire précéder le filtre de sortie d'une règle par une nouvelle racine de persistance.

La règle article_vers_racine_publications ci-dessous illustre cette possibilité en créant une racine `publications` pour nos objets de la classe ***Publication***. Dans cette règle, on peut remarque l'utilisation du regroupement $g(\boldsymbol{\&A})$ qui doit impérativement être en accord avec le paramètre de la fonction de Skolem. La racine est ici peuplée par l'ensemble des références vers les objets articles, tels qu'ils ont été créés dans la règle article_vers_publication2.

rule article_vers_racine_publications =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \stackrel{\star}{\rightarrow} \boldsymbol{A}: \\
\texttt{publications}: & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
\textit{set}\ \{\stackrel{g(\boldsymbol{\&A})}{\rightarrow}\ \boldsymbol{\&Publication}(\boldsymbol{\&A})\ \} & & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
 & & \qquad\qquad \rightarrow \textit{auteurs} \stackrel{I}{\rightarrow} \textit{auteur} \rightarrow Au, \\
 & \Longleftarrow & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
 & & \qquad\qquad \stackrel{\star}{\rightarrow} \_\rangle, \\
 & & An = \mathsf{extraire\_ann\acute{e}e}(Co), \\
 & & An > 1995
\end{array}
$$

Règles paramétriques. Le second mécanisme supporté par YAT$_L$ permet l'écriture de règles qui dépendent de paramètres. Il est ensuite possible d'appler l'une de ces règles à partir d'une autre. Une première utilisation possible de ce mécanisme est de construire un motif à l'aide de deux règles. Nous avons déjà vu comment écrire une règle dépendant d'un paramètre d'entrée. La règle ci-dessous construit un motif qui, pour chaque auteur, contient l'ensemble de ses articles. En l'occurrence, la règle étant paramétrée par le nom d'un chercheur, celui-ci est évalué par rapport à la règle comme une sélection (il n'intervient pas sur le filtrage qui est déterminé par le motif et non pas par la valeur de la variable).

rule les_articles(Au) =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \stackrel{\star}{\rightarrow} \boldsymbol{A}: \\
 & & \quad \textit{article}\langle \rightarrow \textit{numéro} \rightarrow Num, \\
 & & \qquad\qquad \rightarrow \textit{titre} \rightarrow Ti, \\
\textit{set}\{\stackrel{\star}{\rightarrow}\ \boldsymbol{\&Publication}(\boldsymbol{\&A})\ \} & & \qquad\qquad \rightarrow \textit{auteurs} \stackrel{\star}{\rightarrow} \textit{auteur} \rightarrow Au, \\
 & \Longleftarrow & \qquad\qquad \rightarrow \textit{conf} \rightarrow Co, \\
 & & \qquad\qquad \stackrel{\star}{\rightarrow} \_\rangle, \\
 & & An = \mathsf{extraire\_ann\acute{e}e}(Co), \\
 & & An > 1995
\end{array}
$$

Voyons maintenant comme s'effectue l'appel de règle. Il s'agit en fait d'une approche fonctionnelle. Chaque règle article_vers_chercheur′ ci-dessous prend des paramètres en entrées et retourne un motif. Ce motif résultant de l'évaluation est représenté par le nom de la règle auquel on passe les paramètres appropriés. La règle ci-dessous crée le motif chercheur et construit la liste de ses articles grâce à un appel à la règle précédente.

Il s'agit là d'un mécanisme très puissant, autorisant par exemple certaines formes de récursion. Nous verrons d'autres exemples d'utilisation de ce mécanisme dans la suite.

$\underline{\textsf{rule article\_vers\_chercheur}'}$ =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \textit{bibliographie} \xrightarrow{\star} \\
 & & \quad \textit{article}\langle \to \textit{numéro} \to Num, \\
\boldsymbol{Chercheur}(Au): & & \qquad\qquad \to \textit{titre} \to Ti, \\
\textit{class} \to \textit{chercheur} \to \textit{tuple} & & \qquad\qquad \to \textit{auteurs} \xrightarrow{\star} \textit{auteur} \to Au, \\
\quad \{ \to \textit{nom} \to Au, & \Longleftarrow & \qquad\qquad \to \textit{conf} \to Co, \\
\qquad \to \textit{position} \to \text{``''}, & & \qquad\qquad \xrightarrow{\star} \_\rangle, \\
\qquad \to \textit{intérêt} \to \textit{list} & & An = \textsf{extraire\_année}(Co), \\
\qquad \to \textit{articles} \to \texttt{les\_articles}(Au)\} & & An > 1995
\end{array}
$$

5.1.7 Traitement de l'hétérogénéité sémantique

YATL permet d'assurer le traitement de l'hétérogénéité sémantique au niveau structurel. Dans la mesure où l'utilisateur a une connaissance suffisante des sémantiques respectives des sources, il est capable de transmettre cette connaissance dans l'écriture d'un programme d'intégration. Dans ce contexte, la lisibilité des structures d'entrée et de sortie, que nous avions défendue dans le chapitre 4 se justifie pleinement.

Pour illustrer la possibilité d'intégrer des informations issues de plusieurs sources, nous supposons qu'une base de données relationnelle contient des informations sur les chercheurs en base de données d'un laboratoire français. Elle stocke leur nom, leur fonction actuelle et leurs sujets de recherches dans une base avec le schéma suivant.

perso[$\underline{\text{cid: }\texttt{integer}}$, nom: `string`, prénom: `string`, poste: `string`, projet: `integer`]
projet[$\underline{\text{pid: }\texttt{integer}}$, nom: `string`, theme: `string`]

Bien entendu, lorsqu'aucune information sémantique supplémentaire n'est fournies directement au système, la résolution des conflits sémantique demande souvent une analyse préalable par l'utilisateur. Cette analyse concerne à la fois les différences entre les sources et les différences entre les sources et le schéma de la vue intégré (qui peut avoir été conçu indépendamment). Nous voyons que la base de données contient deux relations. La relation perso stocke des informations sur les personnels et fournit le nom et le prénom de la personne ainsi que le poste, qui semble correspondre à l'attribut position dans le schéma objet. La table projet indique un thème de recherche qui devrait fournir une première information quant aux intérêts scientifiques du chercheur.

La règle article_et_pers_vers_chercheur ci-dessous exploite maintenant ces informations pour créer des objets ***Chercheur*** qui contiennent l'ensemble des informations disponibles dans nos deux sources. Elle récupère tout d'abord l'ensemble des informations nécessaires à partir des deux racines de persistance `document_xml` et `perso`. Le problème essentiel ensuite est d'être capable de mettre en relation les informations sur un même chercheur, c'est à dire de trouver le moyen de faire la jointure entre les informations des deux sources. C'est le rôle du prédicat compare_nom qui retourne true lorsque le nom d'un chercheur dans les publications XML (variable Au) coïncide bien avec le nom d'un chercheur dans la base de données relationnelle (dans les variables Nom et Pr).

rule article_et_pers_vers_chercheur =

$$
\begin{array}{lcl}
 & & \texttt{document\_xml}: \\
 & & \mathit{bibliographie} \xrightarrow{\star} \boldsymbol{A}: \\
 & & \quad \mathit{article}\langle \to \mathit{numéro} \to Num, \\
\boldsymbol{Chercheur}(Au): & & \qquad\qquad \to \mathit{titre} \to Ti, \\
\mathit{class} \to \mathit{chercheur} \to \mathit{tuple} & & \qquad\qquad \to \mathit{auteurs} \xrightarrow{\star} \mathit{auteur} \to Au, \\
\quad \{ \to \mathit{nom} \to Au, & \Longleftarrow & \qquad\qquad \to \mathit{conf} \to Co, \\
\quad\quad \to \mathit{position} \to Poste, & & \qquad\qquad \xrightarrow{\star} \_\rangle, \\
\quad\quad \to \mathit{intérêt} \to \mathit{list} & & \\
\quad\quad \to \mathit{articles} \to \mathit{set}\{ & & \texttt{perso}: \\
\quad\quad\quad \xrightarrow{\star} \boldsymbol{\&Publication}(\boldsymbol{\&A}) \ \} \ \} & & \mathit{rel}\{ \xrightarrow{\star} \mathit{row}\{ \to \mathit{cid} \to Cid, \\
 & & \qquad\qquad \to \mathit{nom} \to Nom, \\
 & & \qquad\qquad \to \mathit{prénom} \to Pr, \\
 & & \qquad\qquad \to \mathit{poste} \to Poste, \\
 & & \qquad\qquad \to \mathit{projet} \to Projet\}\}, \\
 & & An = \mathsf{extraire\_année}(Co), \\
 & & An > 1995, \\
 & & \mathsf{compare\_nom}(Au, Nom, Pr)
\end{array}
$$

5.1.8 Programmes génériques

Nous avions vu dans le chapitre 4 que la capacité du modèle à raisonner à différents niveaux de généricité permettait de représenter précisément les informations structurelles des sources. Les filtres de YATL sont conçus sur les motifs du modèle de données, c'est pourquoi ils permettent également d'exploiter la généricité.

L'emploi de filtres génériques dans le langage est utile à plus d'un titre. Cela permet tout d'abord d'écrire une transformation qui s'adapte précisément à la connaissance structurelle de la source. Cela permet également d'écrire des programmes qui fonctionnent sur de larges familles de sources. Cette deuxième possibilité permet d'être moins sensible aux modifications structurelle qui peuvent intervenir dans les sources. Plus un programme est spécifique et plus il y a de risque qu'il doivent être modifié par l'utilisateur en cas de changement. Plus un programme est générique et plus on est sûr qu'il continuera à marcher. Bien entendu, il s'agit là d'un compromis que doit trouver l'utilisateur en fonction des besoins de ses applications.

Nous illustrons maintenant l'écriture d'un programme générique fonctionnant sur toutes les bases ODMG. L'utilisation de la généricité dans un cadre où l'on possède des informations incomplètes sur les structures d'une sources sera quant à elle illustrée dans le prochain chapitre. Le programme ci-dessous, constitué de deux règles convertit toute base de données ODMG en graphe de pages HTML. Il peut être utilisé pour exporter une base de données objet sur la Toile et correspond à la spécification de l'outil O_2Web [O2W96] [3]. La règle objet_vers_page indique qu'un objet est converti en une page HTML. La règle élément convertit

3. Dont l'implantation avec YAT est décrite sommairement dans la section 7.3.2.

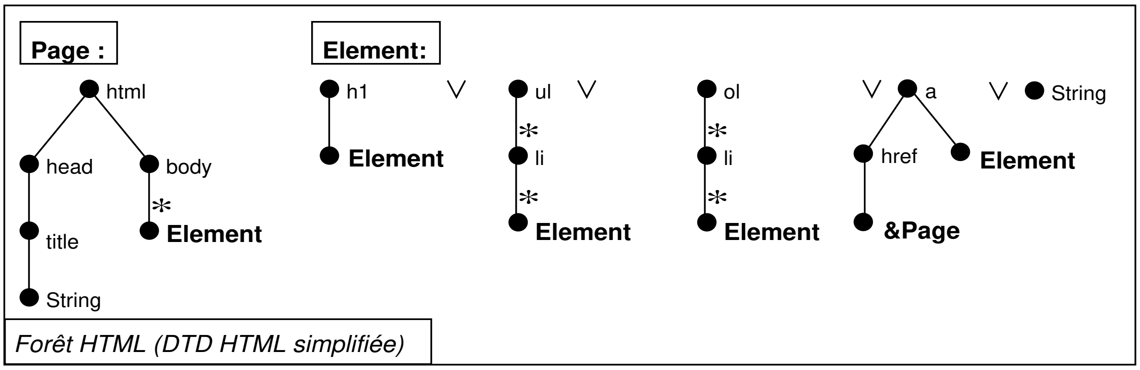


FIG. 5.7: *Description des structures HTML en YAT*

une valeur atomique en chaîne de caractères, une collection ou un n-uplet en liste HTML, et une référence à un objet en une ancre pointant sur la page (l'objet) suivant. Les corps des règles contiennent des filtres correspondant aux différents types ODMG , tandis que la tête crée les motifs HTML correspondants (dont la structure est représentée sur la figure 5.7).

<u>rule page</u>($\boldsymbol{Pclass}$) =

$$\begin{array}{lcl} html\langle \rightarrow head \rightarrow title \rightarrow Classe', & & \boldsymbol{Pclass} : \\ \quad \rightarrow body\langle \rightarrow h1 \rightarrow Classe', & \Longleftarrow & class \rightarrow Classe \rightarrow \boldsymbol{Ptype}.[\boldsymbol{P_1} : \_], \\ \qquad \rightarrow \text{élément}(\boldsymbol{P_1})\rangle\rangle & & Classe' = \text{string\_of\_value}(Classe) \end{array}$$

<u>rule élément</u>($\boldsymbol{P_1}$) =

$$\begin{array}{lcl} S & \Longleftarrow & \boldsymbol{P_1} : Data, \\ & & S = \text{string\_of\_value}(Data) \end{array}$$

$$\begin{array}{lcl} & & \boldsymbol{P_1} : \\ ul \xrightarrow{\star} li\langle \rightarrow Att', & \Longleftarrow & tuple \xrightarrow{\star} Att \rightarrow \boldsymbol{Ptype}.[\boldsymbol{P_2} : \_] \\ \qquad \rightarrow \text{élément}(\boldsymbol{P_2})\rangle & & Att' = \text{concat}(Att, " : ") \end{array}$$

$$\begin{array}{lcl} & \Longleftarrow & \boldsymbol{P_1} : \\ ul \xrightarrow{\star} li \rightarrow \text{élément}(\boldsymbol{P_2})\rangle & & (set \vee bag)\{\xrightarrow{\star} \boldsymbol{Ptype}.[\boldsymbol{P_2} : \_]\} \end{array}$$

$$\begin{array}{lcl} & \Longleftarrow & \boldsymbol{P_1} : \\ ol \xrightarrow{o(I)} li \rightarrow \text{élément}(\boldsymbol{P_2})\rangle & & (list \vee array) \xrightarrow{I} \boldsymbol{Ptype}.[\boldsymbol{P_2} : \_] \end{array}$$

$$\begin{array}{lcl} & & \boldsymbol{P_1} : \&\boldsymbol{Pclass}, \\ a\langle \rightarrow href \rightarrow \&\text{page}(\boldsymbol{Pclass}), & \Longleftarrow & \boldsymbol{Pclass} : \\ \quad \rightarrow Classe'\rangle & & class \rightarrow Classe \rightarrow \boldsymbol{Ptype}.[\boldsymbol{P_2} : \_], \\ & & Classe' = \text{string\_of\_value}(Classe) \end{array}$$

5.2 Typage d'un programme YATL

La conception du langage autour de l'utilisation de filtres et l'existence du système de types pour données semistructurées (que nous avons introduit dans le chapitre précédent) nous mènent naturellement vers l'étude du typage dans YATL. Dans cette section, nous expliquons comment celui-ci est calculé et comment il peut être utilisé. Comme nous l'avons déjà indiqué, un programme YATL prend en entrée une forêt quelconque et retourne une autre forêt (éventuellement vide) en sortie. Notre objectif est de déterminer quel est le type d'un programme YATL, c'est à dire de décrire le plus précisément possible l'ensemble des forêts filtrées par le programme, ainsi que l'ensemble des forêts qui peuvent être générées. Nous donnons les définitions suivantes :

- Le type d'entrée du programme est la plus grande[4] forêt F_{IN} tel que pour toute forêt dans le domaine de F_{IN}, tous les motifs sont filtrés par le programme.
- Le type de sortie du programme est la plus petite forêt F_{OUT} telle que l'ensemble de toutes les forêts générées par le programme est inclus dans le domaine de F_{OUT}.

Nous noterons alors $prg : F_{IN} \mapsto F_{OUT}$ pour indiquer que prg est du type de F_{IN} vers F_{OUT}.

Du fait que les filtres sont conçus sur les expressions du modèle de type, celui-ci peut être aisément inféré à partir du programme. L'inférence de type est réalisée de la façon suivante :

1° on détermine l'ensemble de tous les filtres du programme auxquels on donne un nom ;

2° lorsqu'un nom de motif est passé en paramètre d'une règle à l'autre, on en déduit que les noms sont identiques dans le type.

À titre d'exemple, considérons le programme de la section 5.1.6 constitué des règles rule article_vers_publication2 et article_vers_chercheur. Le type d'entrée du programme est la forêt composée du seul motif ***biographies***. Le type de sortie est défini à l'aide des filtres de sortie, des types des variables d'entrée et des signatures des fonctions externes. Dans notre exemple, le type de sortie est donc la forêt composée des motifs ***Publication*** et ***Chercheur***, avec des restrictions sur les variables An dues à l'utilisation de la fonction externe.

Une fois le type d'un programme déterminé, celui-ci peut être utilisé de plusieurs façons. Il peut être utilisé pour vérifier que l'entrée ou la sortie du programme est bien une instance d'un modèle plus général, que deux programmes peuvent être composés, ou bien encore pour s'assurer que le programme génère bien des motifs dans un type précis. On peut, par exemple, vérifier qu'une transformation génère bien des graphes valides pour un schéma ou pour le modèle ODMG.

Il est important de noter que le typage dans YAT n'est pas contraignant : les programmes n'en ont pas besoin pour fonctionner dans la mesure où un filtrage ne génère jamais d'erreur si le type d'entrée est incompatible. Au pire, le programme ne transforme pas l'information comme on le souhaite.

Indiquons pour finir que l'algorithme de typage présenté ici ne tient pas compte du schéma des sources sur lesquelles le programme est appliqué. Il n'utilise que les informations contenues dans le programme lui-même. Il ne permet donc d'obtenir que le typage le plus général, c'est à dire en supposant que le programme peut être appliqué sur n'importe quelle forêt concrète

4. Au sens de la relation d'instanciation.

d'entrée. Dans le cas où on a plus d'information sur les structures d'entrée, il est alors possible de propager certaines informations de types vers la vue. Cela donne ainsi à l'utilisateur des informations plus précises en vue de l'interrogation. Ce problème est étudié dans [PV98a]. Il pourrait être intéressant de comprendre si certains de ces résultats sont réutilisables dans notre contexte.

5.3 Réutilisation automatique de programmes YAT$_L$

Nous nous intéressons maintenant aux différents mécanismes susceptibles de faciliter la réutilisation de programmes existants. Quelle que soit la simplicité du langage, la plupart des programmeurs préfèrent partir d'un programme existant plutôt que de repartir de zéro. Avec YAT il peut partir d'une transformation générique qu'il a déjà écrite ou fournie par le système et l'adapter facilement à ses besoins.

La première étape était donc de définir des programmes génériques, c'est ce que nous avons fait dans la section 5.1.8. YAT$_L$ supporte plusieurs mécanismes permettant l'exploitation automatique des programmes. Le plus intéressant d'entre eux permet d'instancier automatiquement un programme générique en un programme plus spécifique. Ceci favorise la réutilisation de programmes existants et permet au programmeur de ne pas partir de zéro quand il écrit un programme d'intégration ou de conversion. Il peut réutiliser un programme générique existant (p.ex. celui de la section 5.1.8), puis l'instancier (comme dans la section 5.3.1), le combiner (voir section 5.3.2) ou même le composer (voir section 5.3.3) avec d'autres programmes.

5.3.1 Instanciation

L'une des caractéristiques les plus intéressantes des programmes YAT$_L$ est leur aptitude à être « instanciés ». La possibilité d'étendre l'instanciation aux programmes découle de l'utilisation de filtres fondés sur les motifs qui, eux-mêmes, supportent l'instanciation. Partant d'un programme générique, il peut l'instancier pour que le programme coïncide avec ses données d'entrée.

Supposons par exemple que nous souhaitions afficher les objets de la classe ***Publication*** d'une façon différente de celle proposée par la transformation générique. Réaliser cela avec un outil comme O_2Web nécessite d'écrire du code C++ pour générer les pages correspondantes à la main. Avec YAT, l'utilisateur a non seulement la possibilité d'écrire la transformation qu'il veut de façon déclarative, mais aussi de partir du programme précédent en l'instanciant sur le motif qui l'intéresse pour se simplifier la vie.

Ce procédé est effectué automatiquement pour une règle donnée et un motif instance d'un des filtres de la règle. Le résultat de l'instanciation est un nouveau programme YAT$_L$ dont la sémantique est équivalente à celle du programme générique, mais ayant une signature plus précise. Ce programme peut ensuite être modifié par l'utilisateur comme bon lui semble.

Illustrons ce mécanisme en instanciant le programme de la section 5.1.6 sur la forêt constituée des motifs ***Chercheur*** et ***Publication***. Le résultat de l'instanciation est une nouvelle règle, publication_vers_html, donnée ci-dessous.

Le corps de la règle contient le filtre fourni en entrée du processus d'instanciation et deux appels à la fonction externe string_of_value qui proviennent de l'instanciation de la règle valeur_atomique au niveau du valeur du n-uplet de la classe ***Publication***. Regardons maintenant plus en détail comment cette règle a été dérivée.

1° Tout d'abord la forêt donnée en entrée du processus d'instanciation est appariée avec le type d'entrée du programme. Dans notre exemple, la règle objet_vers_page est mise en relation avec le début du motif ***Publication***, puis la règle nuplet_vers_liste est mise en relation avec le n-uplet, et pour les deux attributs atomiques la règle valeur_atomique, enfin la liste est appariée avec la règle collection_ordonnée.

2° Chacune de ces règles est instanciée indépendemment.

3° Lorsque l'instanciation résulte en une valeur atomique, celle-ci est propagée : soit à travers une fonction externe et/ou directement dans le motif de sortie. Dans notre exemple, c'est ce qui explique l'apparition des chaînes de caractères correspondant aux noms d'attributs et de classe.

4° Les appels à des fonctions externes inutiles sont éliminés ainsi que les filtres dont les variables ne servent pas.

5° Lorsque c'est possible, les règles sont regroupées pour des raisons de lisibilité. Durant cette étape, les conflits entre noms de variables d'une règle à l'autre sont résolus par renommage.

$$
\begin{array}{l}
\underline{\textsf{rule publication\_vers\_html}} = \\
\\
\boldsymbol{Page}(\&\boldsymbol{Publication}) : \\
html \\
\quad \langle \rightarrow head \rightarrow title \rightarrow car, \\
\quad\quad \rightarrow body \langle \rightarrow h1 \rightarrow car, \\
\quad\quad\quad\quad\quad\quad \rightarrow ul \langle \rightarrow li \langle \rightarrow \text{"titre : "}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow Ti_1 \rangle, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow li \langle \rightarrow \text{"année : "}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow An_1 \rangle, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow li \langle \rightarrow \text{"auteurs : "}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow ol \overset{o(I)}{\rightarrow} li \rightarrow a \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \langle \rightarrow href \rightarrow \&\boldsymbol{Page}(\&\boldsymbol{Chercheur}), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow \text{"chercheur"} \rangle\rangle\rangle\rangle\rangle \\
\quad\quad \Longleftarrow \\
\quad\quad\quad\quad\quad \boldsymbol{Publication} : \\
\quad\quad\quad\quad\quad class \rightarrow publication \rightarrow tuple \{ \rightarrow titre \rightarrow Ti, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow année \rightarrow An, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rightarrow auteurs \rightarrow list \overset{I}{\rightarrow} \&\boldsymbol{Chercheur} \}, \\
\quad\quad\quad\quad\quad Ti_1 = \textsf{string\_of\_value}(Ti), \\
\quad\quad\quad\quad\quad An_1 = \textsf{string\_of\_value}(An)
\end{array}
$$

Une fois dérivée, l'utilisateur peut modifier cette règle pour l'adapter à ses besoins. Par exemple, dans la règle suivante, nous avons supprimé l'année de publication et changé le lien pour qu'il indique le nom du chercheur plutôt que le nom de la classe, ce qui est probablement plus utile !

$$
\begin{array}{l}
\underline{\textsf{rule chercheur\_vers\_html\_modifié}} = \\
\\
\boldsymbol{Page}(\&\boldsymbol{Publication}) : \\
html \\
\quad \langle \rightarrow head \rightarrow title \rightarrow car, \\
\quad\quad \rightarrow body\langle \rightarrow h1 \rightarrow car, \\
\qquad\qquad\qquad \rightarrow ul\langle \rightarrow li\langle \rightarrow \text{"titre : "}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow Ti_1\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow An_1\rangle, \\
\qquad\qquad\qquad\qquad\quad \rightarrow li\langle \rightarrow \text{"auteurs : "}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow ol \overset{o(I)}{\rightarrow} li \rightarrow a \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle \rightarrow href \rightarrow \&\boldsymbol{Page}(\&\boldsymbol{Chercheur}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow Nom\rangle\rangle\rangle\rangle\rangle \\
\quad\quad \Longleftarrow \\
\qquad\qquad\quad \boldsymbol{Publication} : \\
\qquad\qquad\quad class \rightarrow publication \rightarrow tuple\{ \rightarrow titre \rightarrow Ti, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \textit{année} \rightarrow An, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow auteurs \rightarrow list \overset{I}{\rightarrow} \&\boldsymbol{Chercheur}\}, \\
\qquad\qquad\quad \boldsymbol{Chercheur} : \\
\qquad\qquad\quad class \rightarrow chercheur \rightarrow tuple\{ \rightarrow nom \rightarrow Nom, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow position \rightarrow Pos, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \textit{intérêt} \rightarrow \_, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow articles \rightarrow \_\}, \\
\qquad\qquad\quad Ti_1 = \textsf{string\_of\_value}(Ti), \\
\qquad\qquad\quad An_1 = \textsf{string\_of\_value}(An)
\end{array}
$$

5.3.2 Combinaison de programmes

L'instanciation permet donc de déduire un programme de conversion spécifique à partir d'un autre plus générique. Dans de nombreux exemples pratiques, on souhaite juste avoir une transformation spécifique pour une certaine catégorie de motifs, tandis que l'ensemble des autres reste traité par la conversion générique. Par exemple, on peut vouloir utiliser la règle publication_vers_html_modifié ci-dessus pour les publications, tandis que tous les autres objets continueront à être convertis de façon générique. Cela permet également de donner une réponse satisfaisante aux problème d'évolution : si le schéma pour les publications change, alors elle seront de nouveau converties de façon générique. Pour obtenir ce résultat, une première idée est d'introduire la règle spécifique dans le programme générique. Cependant, effectuer cette opération sans précaution est dangereux car le programme propose alors deux conversions distinctes pour les publications.

Pour résoudre ce problème, l'interprète YAT$_{\text{L}}$ organise l'ensemble des règles hiérarchiquement en se basant sur la relation d'instanciation. La politique par défaut est alors la suivante : lorsqu'il y a conflit entre deux règles, c'est la règle la plus spécifique qui l'emporte.

Un conflit entre deux règles apparaît quand les conditions suivantes sont réunies : (i) elles codent potentiellement pour le même motif de sortie (*i.e.* elles utilisent la même fonction de Skolem) et (ii) lorsque leurs filtres d'entrées sont instances l'un de l'autre. La première condition est indispensable pour pouvoir utiliser le même motif dans plusieurs règles. Notons que l'utilisateur peut imposer une certaine hiérarchie pour les besoins d'applications particulières. Bien entendu, cela transgresse la politique déclarative du langage.

5.3.3 Composition

Le dernier mécanisme que nous décrirons consiste à composer deux programmes automatiquement. Pour les besoins de l'explication, nous imaginons que, partant des articles XML, nous ne souhaitons plus charger une base de donnée à objet mais directement exporter les informations vers la toile. Habituellement, cela nécessiterait de tout réécrire à partir de zéro. Pourtant, nous avons déjà à notre disposition deux programmes, l'un de XML vers le modèle ODMG, et l'autre de l'ODMG vers HTML. Une première possibilité est donc, partant du document XML de départ, d'appliquer successivement ces deux programmes. Cela a cependant quelques inconvénients. Tout d'abord, avec deux programmes successifs, il peut devenir difficile à l'utilisateur de saisir la sémantique de la transformation et de faire des modifications. Ensuite, cela pénalise inévitablement les performances lors de l'exécution. Le mécanisme de composition nous donne l'opportunité de faire mieux, en générant automatiquement un programme transformant directement le document XML en HTML.

Le principe de la composition est le suivant. Partant de deux programmes YATL dont les signatures sont $prg1 : F_1 \mapsto F_2$ et $prg2 : F'_2 \mapsto F_3$, le système vérifie tout d'abord que F_2 est une instance de F'_2 [5]. Si c'est le cas, le système instancie $prg2$ avec la forêt F_2, ce qui produit une nouvelle transformation $prg2' : F_2 \mapsto F'_3$. Ensuite, la composition finale a lieu. Celle-ci devient triviale car les filtres de sortie de la première transformation et les filtres d'entrée de la seconde sont syntaxiquement équivalents.

Dans notre exemple, nous voulons composer la transformation de XML vers ODMG avec la transformation générique de ODMG vers HTML. La première étape consiste à instancier le programme de ODMG vers HTML de la même façon que dans la section 5.3.1. Ensuite, ce programme est composé avec le programme de XML vers ODMG. On en déduit deux nouvelles règles. Celle qui crée les pages HTML pour les publications est donnée ci-dessous.

```
rule article_directement_vers_html =

Page(&A) :
html                                                  document_xml :
 ⟨ → head → title → car,                              bibliographie *→ A :
   → body⟨ → h1 → car,                                   article⟨ → numéro → Num,
          → ul⟨ → li⟨ → "titre: ",                                → titre → Ti,
                      → Ti1⟩,                ⟸                    → auteurs I→ auteur → Au,
                 → li⟨ → "année: ",                               → conf → Co,
                      → An1⟩,                                     *→ _⟩,
                 → li⟨ → "auteurs: ",                 An = extraire_année(Co),
                      → ol o(I)→ li → a               Ti1 = string_of_value(Ti),
                       ⟨ → href → &Page(Au),          An1 = string_of_value(An)
                         → "chercheur"⟩⟩⟩⟩⟩
```

5. Ce qui correspond à la relation de contravariance utilisée pour vérifier que l'on peut composer deux fonctions.

5.4 Filtrage précis vs filtrage partiel dans YATL

Dans la plupart des autres langages, la sémantique du filtrage est *partielle*. C'est à dire qu'une donnée d'entrée est filtrée à partir du moment où elle fournit *au moins* les « champs » requis par le filtre. Il n'est ainsi pas nécessaire de connaître exactement la structure pour interroger les données.

À l'inverse, le filtrage dans YATL est *total* dans la mesure où il est nécessaire que la donnée d'entrée soit exactement une instance du filtre pour que le filtrage ait lieu. Cependant, nous avons vu que la puissance du système de types et sa flexibilité permettent de jouer sur la généricité pour manipuler des données dont on connaît partiellement la structure.

Ainsi, le filtre suivant :

$$
\begin{array}{ll}
f_1 = & \\
article\langle & \xrightarrow{\star} \boldsymbol{Yat}, \\
& \rightarrow numéro \rightarrow Num, \\
& \xrightarrow{\star} \boldsymbol{Yat}, \\
& \rightarrow titre \rightarrow Ti, \\
& \xrightarrow{\star} \boldsymbol{Yat}, \\
& \rightarrow résumé \rightarrow Res, \\
& \xrightarrow{\star} \boldsymbol{Yat}\rangle
\end{array}
$$

permet de filtrer tous les motifs qui ont *au moins* les champs numéro, titre et résumé. En effet, les branches $\xrightarrow{\star} \boldsymbol{Yat}$ capturent toutes les branches alternatives possibles. On voit clairement comment le pouvoir d'expression des motifs YAT suffit pour exprimer un filtrage partiel.

Notons que, dans certains cas, il peut s'avérer délicat de fabriquer des branches additionnelles qui évitent les ambiguïtés lors du filtrage. Heureusement, la fermeture du modèle de type par rapport aux opérations ensemblistes, notamment la complémentation, assurent qu'un filtre partiel peut toujours être construit à partir d'un filtre total.

Bien entendu, le motif ci-dessus possède deux inconvénients. D'abord, il n'est pas très lisible et il serait difficile à écrire pour l'utilisateur. Ensuite, appliquer le filtrage sur de tels filtres peut s'avérer coûteux. Pour répondre au premier problème, nous introduisons une nouvelle notation : $\langle| \ldots |\rangle$. Le filtre ci-dessous est identique au filtre précédent mais adopte cette nouvelle notation pour indiquer un filtrage partiel.

$$
\begin{array}{ll}
f_2 = & \\
article\langle| & \rightarrow numéro \rightarrow Num, \\
& \rightarrow titre \rightarrow Ti, \\
& \rightarrow résumé \rightarrow Res|\rangle
\end{array}
$$

Une évaluation naïve pour le filtrage partiel est de construire automatiquement le filtre f_1 à partir du filtre f_2, puis d'appliquer l'opération de filtrage habituelle. Les filtres complets peuvent rapidement devenir très complexe, ce qui pose la question des performances. Il serait donc intéressant d'offrir un algorithme d'évaluation particulier pour le filtrage partiel (cela correspondrait à deux implantations de l'opération de *Bind* que nous allons voir dans le chapitre 6).

Dans le prochain chapitre, nous serons amenés à formuler des *requêtes* sur la vue intégrée pour illustrer leur évaluation. Pour nous rapprocher d'un langage de requête sans pour autant introduire de nouveaux concepts, nous utiliserons YAT$_L$ avec les notations et la sémantique du filtrage partiel.

Chapitre 6

Évaluation et optimisation

> 'A slow sort of country!' said the Queen. 'Now, *here*, you see, it takes all the running *you* can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!'
>
> – Lewis CARROLL
>
> (in *Through the Looking Glass*)

NOUS ÉTUDIONS DANS CE CHAPITRE LES MÉCANISMES liés à l'évaluation de programmes d'intégration ou de requêtes sur une vue intégrée. La quantité de données à traiter lors de l'intégration est souvent importante. C'est pourquoi il est essentiel de mettre au point des techniques efficaces pour l'évaluation des programmes d'intégration tels qu'ils ont été présentés dans le chapitre précédent. Jusqu'à présent, seuls les systèmes structurés fournissaient les performances nécessaires pour l'intégration de sources volumineuses. Nous proposons une approche permettant de combler les lacunes des systèmes semistructurés dans ce domaine en exploitant les opportunités d'optimisation introduites par le système de types décrit au chapitre 4.

Dans la section 6.2, nous introduisons le modèle opérationnel, l'algèbre pour données semistructurées ainsi que les équivalences algébriques vérifiées par celle-ci. Enfin, nous montrons comment transcrire un programme ou une requête YATL dans l'algèbre. Dans la section 6.3, nous montrons comment il est possible d'exprimer précisément les capacités opérationnelles de sources hétérogènes, puis d'exploiter ces informations pour mettre au point des techniques d'optimisation distribuées. Mais auparavant, nous décrivons, dans la section 6.1, un scénario impliquant l'intégration de sources culturelles qui servira de support à la présentation. Les sources sont choisies pour leur hétérogénéité tant du point de vue des structures de données que des opérations disponibles. L'objectif est ainsi d'illustrer la capacité du système à s'adapter aux différents cas de figure, du plus structuré au semistructuré.

6.1 Scénario d'une application d'intégration

Afin d'illustrer les techniques qui vont être proposées dans ce chapitre, nous mettons en place un scénario adapté. Nous souhaitons procéder à l'intégration de deux sources de données dans le cadre d'une application culturelle, inspirée du projet Européen AQUARELLE[1].

1. `http://aqua.inria.fr/`

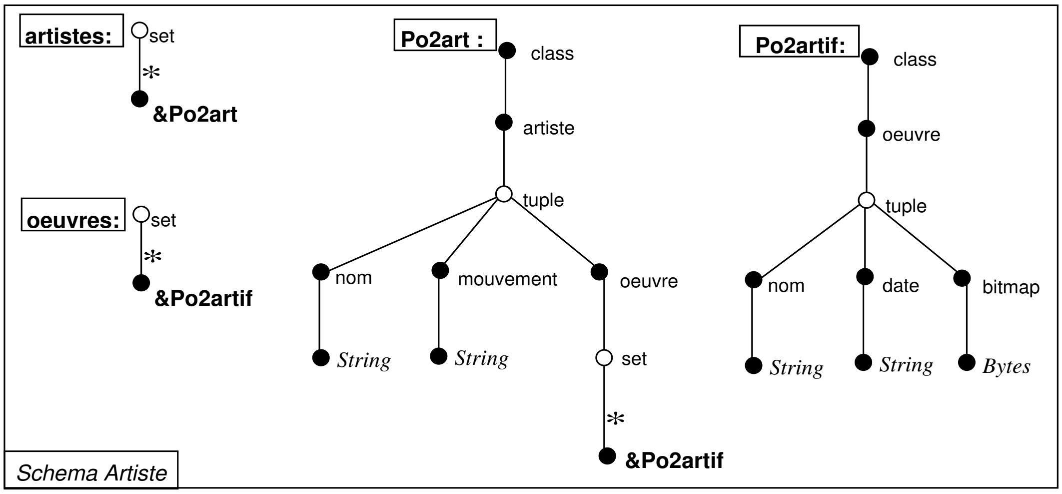


FIG. 6.1: *Schéma O_2 sur les artistes modernes*

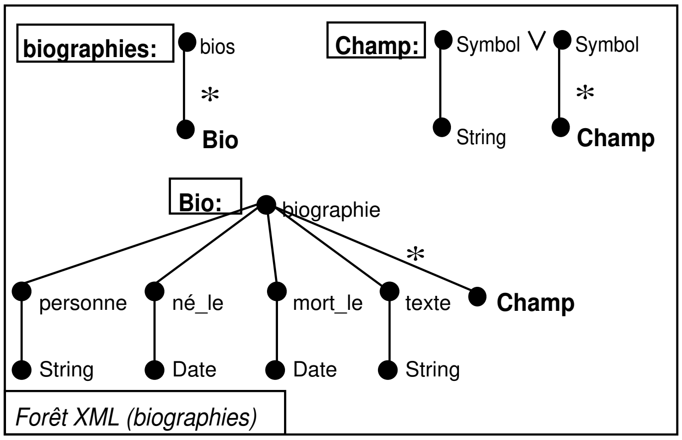


FIG. 6.2: *Structure des données biographiques XML*

Présentation des sources. La première source est une base de données O_2 [BDK92] qui contient des informations sur les artistes modernes et sur leur œuvre. La représentation YAT du schéma correspondant est donnée sur la figure 6.1. Cette source peut évaluer des requêtes OQL quelconques sur les données qu'elle contient, et chaque classe définit des méthodes qui lui sont spécifiques.

La seconde source est une base de données XML qui contient des biographies d'artistes. Comme on peut le voir sur la figure 6.2, les documents contenus dans la base ont des champs obligatoires, dont l'un d'entre eux (texte) est tiré d'un dictionnaire culturel et un nombre arbitraire de champs additionnels (décrits par le motif ***Champ***). Les biographies sont indexées avec Wais [Pfe95]. Deux index permettent de répondre à des requêtes soit sur le contenu textuel global des documents (en utilisant le prédicat contains), soit sur le contenu du champ texte (en utilisant le prédicat text_contains).

Description de la vue intégrée. Nous voulons offrir une vue homogène de ces deux sources. Nous décrivons donc l'intégration par un programme YAT_L. Celui-ci est composé des deux règles données ci-dessous.

La première règle rule odmg_xml_vers_artistes génère un ensemble de motifs ***&Artiste***(N), chacun contenant l'ensemble des informations concernant un artiste donné, et peuple la racine de persistance `tous_les_artistes`. Ces informations sont tirées de la base de données à objets et des documents XML : on peut remarquer l'utilisation de la même variable N dans les filtres d'entrées ***Bio*** et ***A*** pour réaliser l'association entre les informations des deux sources (jointure). Cette variable est donc naturellement utilisée comme paramètre de la fonction de Skolem en sortie. Notons aussi la création de références vers les motifs créés dans la seconde règle, grâce à l'appel de la fonction de Skolem ***&Oeuvre***(N_A, D_A). Enfin, le motif ***C*** sert à exporter les champs additionnels dans la vue, afin de permettre à l'utilisateur d'en prendre connaissance et de les interroger.

La seconde règle est, quant à elle, très simple, créant un motif pour chaque œuvre contenue dans la base.

rule odmg_xml_vers_artistes =

`tous_les_artistes :`
$root \xrightarrow{\star}$ ***&Artiste***(N) :
$artiste\langle \rightarrow nom \rightarrow N,$
$\rightarrow né\_le \rightarrow B,$
$\rightarrow mort\_le \rightarrow D,$
$\rightarrow mouvement \rightarrow S,$
$\rightarrow œuvres \xrightarrow{\star}$ ***&Oeuvre***$(N_A, D_A),$
$\rightarrow description \rightarrow T,$
$\rightarrow biographie \xrightarrow{\star}$ ***C*** : $\rangle$
	$\Leftarrow$	`artistes :`
		$set\{\xrightarrow{\star}$ ***&A***$\}$,
		A :
		$class \rightarrow artiste \rightarrow tuple$
		$\{ \rightarrow nom \rightarrow N,$
		$\rightarrow mouvement \rightarrow S,$
		$\rightarrow œuvres \rightarrow set\{\xrightarrow{\star}$ ***&O***$\}\}$,
		O :
		$class \rightarrow œuvre \rightarrow tuple\{ \rightarrow nom \rightarrow N_A,$
		$\rightarrow date \rightarrow D_A\},$
		`biographies :`
		$bios \xrightarrow{\star}$ ***Bio*** :$biographie\langle \rightarrow personne \rightarrow N,$
		$\rightarrow né\_le \rightarrow B,$
		$\rightarrow mort\_le \rightarrow D,$
		$\rightarrow texte \rightarrow T,$
		$\xrightarrow{\star}$ ***C*** : $\rangle$

rule odmg_vers_œuvres =

`toutes_les_œuvres :` $root \xrightarrow{\star}$ ***&Oeuvre***(N, D): $œuvre\langle \rightarrow nom \rightarrow N,$ $\rightarrow date \rightarrow D$ $\rightarrow image \rightarrow B\rangle$	$\Leftarrow$	`œuvres :` $set\{\xrightarrow{\star}$ ***&O*** $\}$, ***O*** : $class \rightarrow œuvre \rightarrow tuple\{ \rightarrow nom \rightarrow N,$ $\rightarrow date \rightarrow D,$ $\rightarrow bitmap \rightarrow B\}$

Exporter une opération vers la vue. Une fonctionnalité additionnelle dans ce contexte où l'on s'intéresse aux caractéristiques opérationnelles du système est la possibilité de rajouter des opérations dans la vue. Lorsque l'on définit une vue en YAT, il est possible d'exporter simultanément une opération définie sur les informations de la vue en fonction d'opération définie sur les objets des sources. La règle ci-dessous informe le système de l'existence d'une nouvelle opération (description_contains) qui prend en entrée un artiste, et explique comment une telle opération est définie en fonction des opérations sur les données sources qui lui correspondent.

view operation on rule odmg_xml_vers_artistes =

description_contains(X: *String*, ***Artiste***(N)) $\equiv$ text_contains(X: *String*, ***Bio***)

Notons que ce mécanisme est tout à fait similaire à celui qui consiste à exporter des méthodes dans le cadre des vues objet [SAD94, GBCG97, VW98].

Cet exemple sera utilisé tout au long du chapitre. Mais pour l'heure, commençons par introduire le modèle opérationnel.

6.2 Modèle opérationnel

Dans cette section, nous présentons le modèle opérationnel fourni en standard par YAT, c'est à dire l'ensemble des opérations disponibles ainsi que la façon dont elles peuvent être exploitées. Il s'agit de l'ensemble des opérations de bases supportées par le médiateur, car c'est lui dans un premier temps qui doit réaliser l'évaluation des requêtes et du programme YATL.

6.2.1 Introduction : une approche fonctionnelle

Nous prenons le parti d'un modèle opérationnel de nature fonctionnelle. C'est à dire que le modèle est constitué d'un ensemble de fonctions, avec chacune une signature et une sémantique précises. Ces fonctions peuvent être composées arbitrairement dans la mesure où leurs signatures correspondent. Certaines de ces fonctions sont des opérateurs algébriques. Ils jouent un rôle essentiel dans la mesure où ils permettent d'exprimer l'évaluation d'un langage de manipulation semistructuré, tel YATL. Par ailleurs, le choix de ces opérateurs algébriques est fait de telle sorte qu'ils vérifient un certain nombre d'équivalences. Ces équivalences sont les fondements sur lesquels reposent les différentes techniques d'optimisation que nous présentons dans la suite.

Afin de clarifier la présentation du modèle opérationnel, nous l'organisons comme un graphe sur lequel nous classons les opérations disponibles. Ce graphe est illustré sur la figure 6.3. On peut y voir les opérateurs de l'algèbre à droite et les autres fonctions à gauche. Les fonctions booléennes sont regroupées sous la dénomination Prédicat : elles seront utilisées par certains opérateurs algébriques (notamment les opérations de sélection).

Le graphe donne l'ensemble des opérations que connaît le médiateur et pour chacune d'elle, l'ensemble des systèmes qui sont en mesure de l'évaluer. Chaque opération possède une signature et une sémantique, dont on peut consulter la description. Avant toute connection d'un adaptateur, cet ensemble est restreint aux seules opérations du système YAT (*i.e.* supportées par le médiateur). Nous verrons dans la section 6.3.1 comment cet ensemble est enrichi par les opérations des sources.

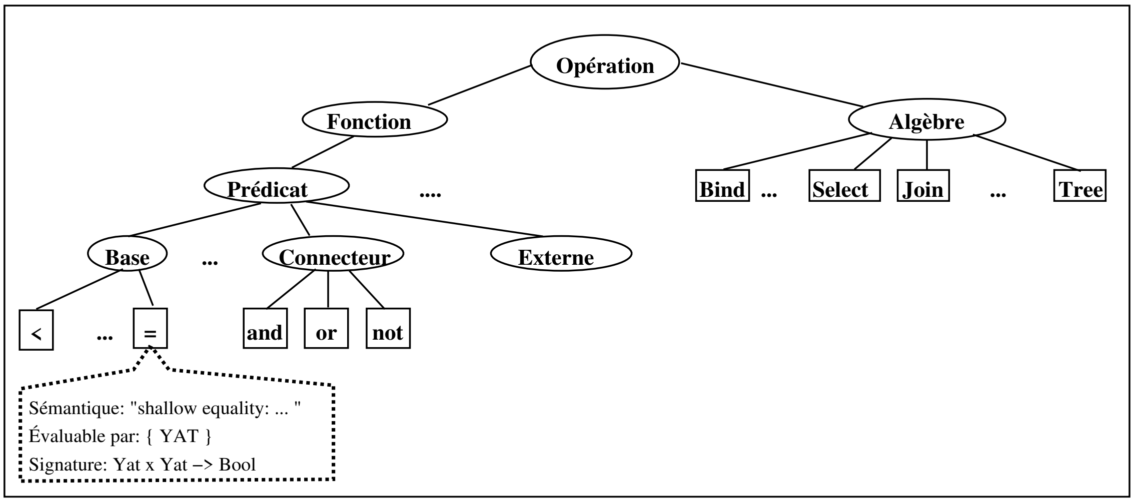


FIG. 6.3: *Organisation des opérations*

6.2.2 L'algèbre

Principe. Le cœur du modèle opérationnel est constitué d'une algèbre pour données semi-structurées. La principale difficulté lors de l'élaboration d'une algèbre est d'obtenir de bonnes propriétés pour l'optimisation. Cela peut s'avérer difficile lorsque, comme c'est le cas ici, des structures de données arbitraires sont manipulées.

Cependant, on peut remarquer que le modèle de données de YAT partage des points communs avec les modèles objets, dans le sens où il offre des structures arbitrairement imbriquées (ici de nature arborescente), et une notion de référence similaire. C'est pourquoi nous adoptons une technique inspirée de celle utilisée dans les algèbres objet. Partant d'une structure de données YAT quelconque, nous appliquons tout d'abord un opérateur dont le rôle est d'extraire les informations nécessaires. Cet opérateur est appelé *Bind* et produit une structure régulière baptisée *Tab*, qui s'apparente à une collection de n-uplets non atomiques, telle qu'on en trouve dans les modèle objets. Sur ces structures *Tab*, nous pouvons de nouveau appliquer un ensemble d'opérateurs standards, tels que la sélection (*Select*), la projection (*Project*), la jointure (*Join*), etc. Pour finir, une opération complémentaire du *Bind*, dénotée par l'opérateur *Tree*, produit une nouvelle structure YAT quelconque comme résultat.

Nous présentons successivement l'opérateur *Bind* et la structure *Tab*, l'opérateur *Tree*, et enfin nous passons en revue les autres opérateurs de l'algèbre.

L'opérateur Bind. D'une part, cet opérateur sert à extraire les données utiles à partir d'un motif quelconque, par rapport à un certain filtre. D'autre part, il produit une représentation des variables et des valeurs qui leur sont liées sous forme de tableau. Nous verrons par la suite comment cet opérateur peut être utilisé pour capturer les propriétés de filtrage et de liaison de variables des langages de requêtes externes. La figure 6.4 illustre sur un exemple simple l'utilisation du Bind.

Le Bind est ici appliqué sur la racine de persistance ***artistes***, qui représente la collection d'artistes de notre base O_2. Le filtre permet l'extraction pour chaque artiste de son nom dans la variable $\$N$ (notez que dans ce chapitre, nous utiliserons le symbole \$ dans les filtres du *Bind* pour indiquer une variable) et du mouvement artistique auquel il appartient ($\$M$). La

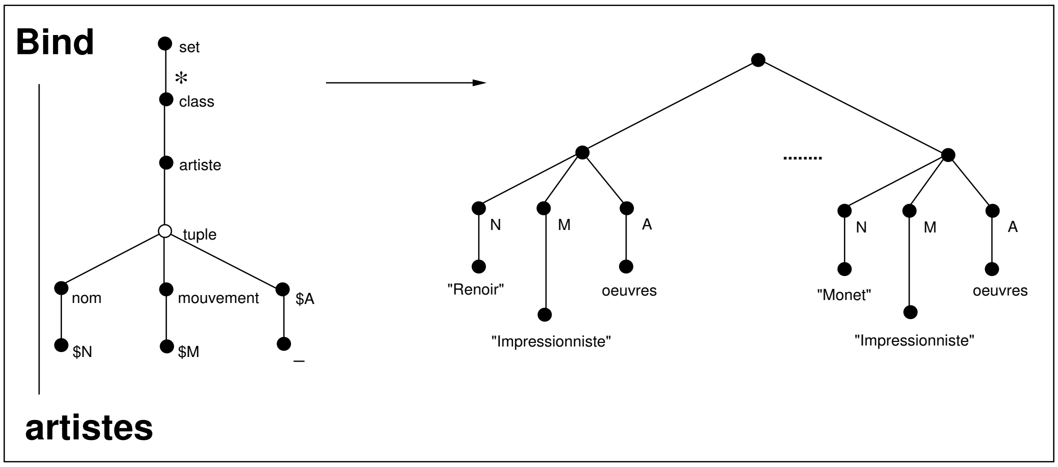


FIG. 6.4: *Opérateur Bind*

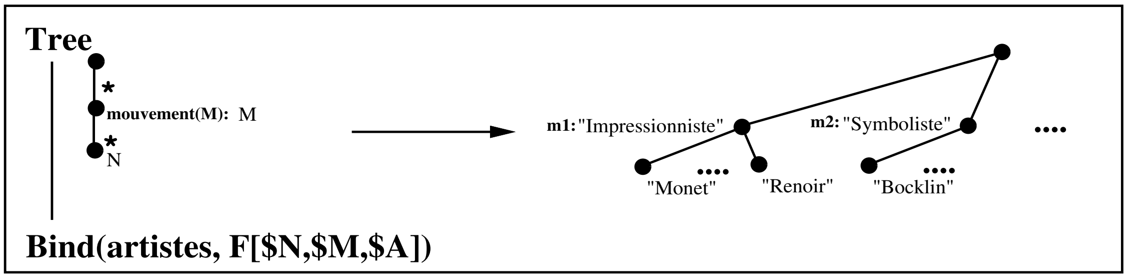


FIG. 6.5: *Opérateur Tree*

variable supplémentaire $\$A$ représente le nom du dernier attribut. Le résultat de l'opération est montré à droite de la figure 6.4 : il s'agit d'un motif ayant un fils pour chaque liaison valide pour les trois variables $N - P - A$.

L'opérateur Bind permet donc de retrouver une certaine régularité, que pourront exploiter les autres opérateurs de l'algèbre qui sont similaires à ceux des algèbres relationnelles ou objet. Notons que cet opérateur est très similaire aux opérateurs utilisés dans d'autres algèbres (*Replace* dans [AB95], *Image* dans [SZ89], χ dans [KM94] or *Map* dans [CM93]) ou encore au Scan dans [CM96].

L'opérateur Tree. Cet opérateur est appliqué sur une structure de Tab et effectue l'opération contraire. Il crée un nouveau motif concret à partir de ce Tab et d'un filtre de sortie. Sur la figure 6.5, le Tree est appliqué sur le résultat du Tab de la figure précédente (dénoté par l'expression $F[\$N, \$P, \$A]$). Le filtre de sortie crée ici une collection et utilise la fonction de Skolem ***mouvement*** pour indiquer la création d'un indentifiant pour chaque P distinct (ce qui implique par là même un regroupement sur cette même variable). Les motifs correspondants sont annotés par M à la racine, puis ont un ensemble de fils, un pour chaque liaison de N.

Reste de l'algèbre. Les autres opérateurs sont ceux de l'algèbre objet de Cluet et Moerkotte [CM93] : *Map*, *Join*, *DJoin* (jointure dépendante, équivalente à une boucle imbriquée),

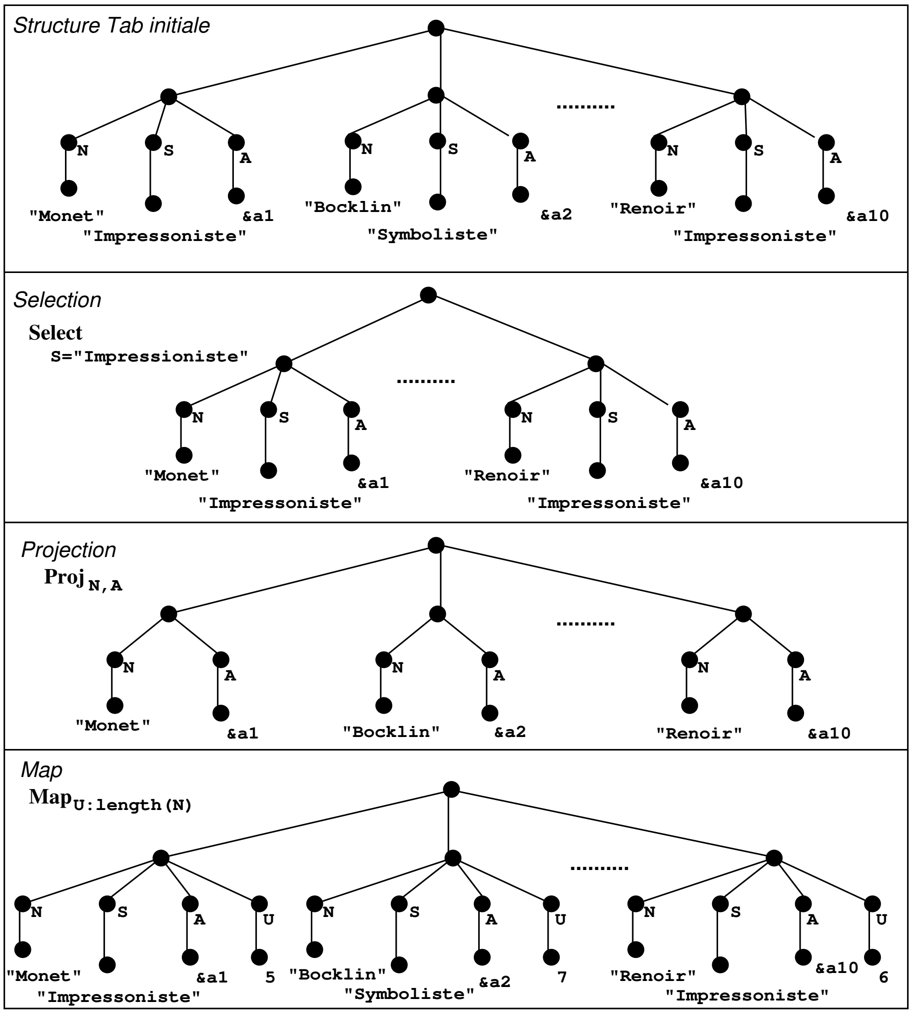


FIG. 6.6: *Autres opérateurs*

Select, *Project*, *Group*, *Sort*, *Union* et *Intersection*. Tous ces opérateurs sont définis sur des structures de Tab (voir la section 6.5 pour une définition formelle de ces opérations). La similitude entre Tab et collections de n-uplets permet de conserver entre ces opérteurs toutes les équivalences connues dans le contexte objet. La figure 6.6 illustre la transposition de quelques uns de ces opérateurs dans YAT.

6.2.3 Algébraisation de YATL

Dans cette section, nous montrons comment transformer un programme YATL en un ensemble d'expressions algébriques équivalentes. Chaque règle est convertie en une expression algébrique de la façon suivante.

1° Une racine de persistance correspond à une opération évaluable.

2° Chaque filtre d'entrée est converti en un *Bind*, opération qui capture précisément la sémantique du filtrage. Cela permet de créer la structure Tab qui permet une évaluation ultérieure par les autres opérateurs de l'algèbre.

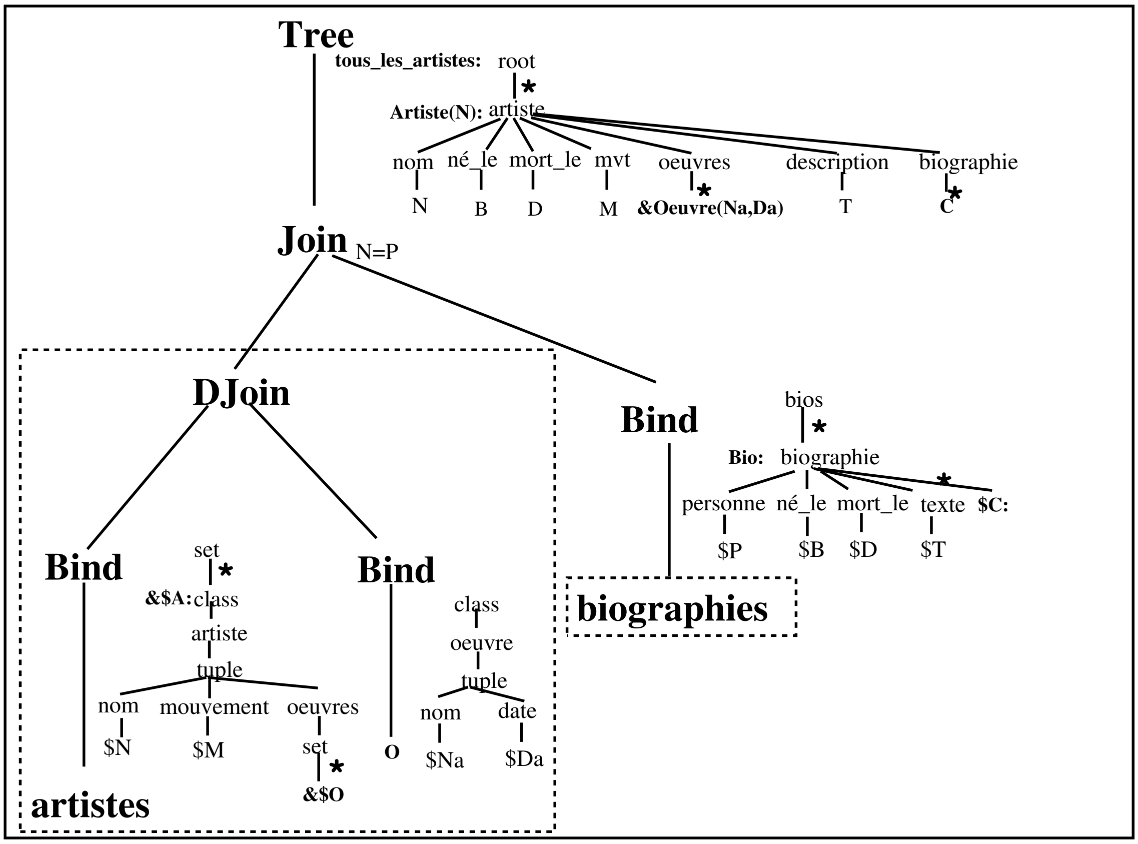


FIG. 6.7: *Algébraisation de la règle* rule odmg_xml_vers_artistes

3° La connection entre les différents filtres d'entrées est capturée comme une opération de jointure. Il s'agit d'une jointure traditionnelle (*Join*) pour une relation simple définie par un prédicat (p.ex. $N = P$ entre les filtres ***Bio*** et ***A***) ou un *Djoin* pour la navigation (p.ex. motif ***O***).

4° Les prédicats et fonctions externes sont convertis respectivement en opérations *Select* et *Map*.

5° Enfin, le motif de sortie est converti en un *Tree*.

Partant de la règle rule odmg_xml_vers_artistes de notre scénario, le résultat de l'application de ces quelques étapes très simples est illustré sur la figure 6.7. On retrouve les deux racines de persistance `artistes` et `biographies`, ainsi que des *Bind* correspondant aux filtres d'entrées (notez que, du côté objet, nous avons concaténé deux *Bind* et le *Djoin* sur la variable ***O*** en un seul *Bind* pour réduire la taille de l'expresion). La relation sur la variable N est représentée par une jointure, tandis que la navigation par ***O*** est représentée par un *Djoin*. Pour finir, une opération *Tree* construit le résultat.

Dans le cas où l'on souhaite matérialiser la vue, c'est bien cette expression qui sera évaluée. On peut envisager de l'optimiser, même si dans le cas présent, il n'y a pas grand chose à faire. Puisque nous sommes dans un contexte impliquant des sources distribuées, on peut cependant envisager de déporter une partie de l'évaluation vers les sources. Nous verrons comment dans la section suivante.

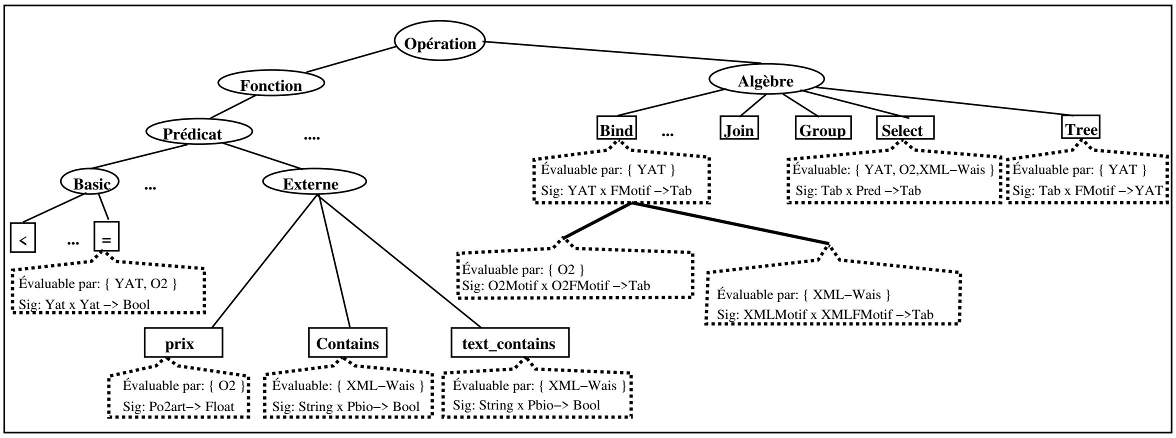


FIG. 6.8: *Ajout des capacités d'O_2 et de la source XML-Wais*

6.3 Techniques d'optimisation distribuées

6.3.1 Description des capacités des sources

Exploiter les capacités des sources dans le traitement des requêtes est une des techniques les plus importantes dans un contexte distribué. La possibilité de pousser l'évaluation de certaines requêtes vers des systèmes sources externes permet : de réduire le temps de traitement de ces requêtes en utilisant des index spécifiques aux sources ; de minimiser aussi bien les coûts de communication entre les systèmes sources et le médiateur que les coûts de conversion dans le modèle pivot ; de limiter les ressources système que nécessite le médiateur (p.ex. la mémoire utilisée) ; et enfin de bénéficier des possibilités de parallélisation.

Pour toutes ces raisons, la qualité des informations que peut fournir une source à propos de ses propres capacités de traitement devient un point important qui mérite d'être étudié. Afin de représenter ces informations le plus rigoureusement possible, le modèle opérationnel YAT offre deux caractéristiques principales : (i) il est extensible et (ii) il peut représenter des capacités de requêtes génériques. L'extensibilité du modèle opérationnel sert à remédier au fait qu'il est impossible d'imaginer à l'avance toutes les opérations qu'une source externe peut proposer. Par ailleurs, la possibilité de représenter des capacités génériques permet de décrire des langages aussi complexes que SQL ou OQL. Ces deux caractéristiques sont décrites respectivement dans les sections 6.3.2 et 6.3.3. Enfin, le paragraphe 6.3.4 explique comment déterminer précisément quelle partie des traitements peuvent être poussés sur les systèmes sources.

Avant de commencer, la figure 6.8 reprend le graphe de la figure 6.3 et montre une représentation partielle du modèle opérationnel YAT après l'intégration des capacités des deux sources de notre exemple. Sur cette figure, la spécialisation est indiquée par un trait gras. Comme nous l'avions déjà indiqué, lorsqu'aucun adaptateur n'est connecté, les opérations répertoriées sont les opérations de base du modèle opérationnel et seul médiateur YAT est capable de les exécuter. Dans la suite, nous expliquons comment chaque adaptateur connecté exporte de nouvelles opérations vers le médiateur. Il peut également fournir les équivalences correspondantes, permettant ainsi d'intégrer cette nouvelle opération dans l'optimisation. Enfin, notons que nous prêterons attention aux aspects liés à la généricité des adaptateurs, dont on a vu qu'il s'agissait d'une caractéristique essentielle.

6.3.2 Extensibilité

Trois arguments justifient la possibilité d'étendre le modèle opérationnel de base par de nouvelles opérations. Le premier est que cela permet d'offrir des possibilités d'interrogation plus variées à l'utilisateur. Le deuxième est que certaines requêtes ne peuvent être évaluées qu'en utilisant ces nouvelles opérations. Enfin, le troisième argument est que l'addition de nouvelles opérations peut apporter des possibilités d'optimisation supplémentaires. Nous étudions ces cas plus en détail.

Extensions fonctionnelles simples. La nature fonctionnelle du modèle opérationnel YAT autorise toute composition d'opérations dont le type est consistant. Pour cette raison, il peut être facilement étendu avec de nouvelles fonctions du premier ordre. Un adaptateur peut signaler au médiateur la présence d'une nouvelle fonction en indiquant sa signature. Cette fonction peut alors être utilisée dans la définition d'une vue ou dans une requête, dans la mesure, bien entendu, que son utilisation soit bien typée. Ce type d'extension simple est illustré sur la figure 6.8 par l'ajout de la méthode `prix` fournie dans la classe O_2. La signature de cette méthode est donnée à l'aide des types YAT importés précédemment (dans ce cas particulier, cela est fait au niveau du schéma en utilisant le motif $Po2art$). Notons que l'exportation de toutes les méthodes peut être faite de façon générique par l'adaptateur O_2 en interrogeant le gestionnaire de schémas du système O_2.

Utilisation de sources de données non-standard. Dans les bases de données, et dans beaucoup d'autres types de sources, l'opération la plus élémentaire consiste à accéder à un point d'entrée de la base ou de la source (p.ex. une relation, une racine de persistance, la racine d'un système de fichier). De façon surprenante, cette opération sensément simple n'est pas toujours possible dans certaines sources. Par exemple, beaucoup de sites sur la Toile ne sont accessibles qu'à travers des formulaires et n'exportent pas l'intégralité de leur contenu (p.ex. AltaVista). Dans ces conditions, il est capital de comprendre quelles opérations sont supportées par ces sources même si ces opérations n'ont pas de contrepartie dans le modèle opérationnel de base. En effet, seule la connaissance de ces opérations nous permettra d'accéder aux sources en question et donc de les intégrer.

L'accès à de telles sources se fait en deux temps dans YAT. Tout d'abord, les prédicats ou opérations supportées par ces sources (p.ex. requêtes plein-texte, etc) sont importées dans le modèle du médiateur. Ensuite, l'algèbre est spécialisée pour capturer la façon dont ces prédicats peuvent être utilisés. Nous verrons comment la spécialisation s'opère dans la section 6.3.3. Pour le moment, observons juste sur la figure 6.8 que dans le cas de notre source XML-Wais, les deux prédicats textuels accessibles ont été ajoutés : contains sur le contenu complet des documents et text_contains qui est spécifique au champ texte.

Une autre hypothèse courante est que l'on ne peut interroger que ce que l'on peut voir ! Une fois de plus, cela n'est pas toujours le cas, par exemple dans le protocole Z39.50 [ANS95] (particulièrement utilisé dans le domaine des librairies numériques et qui sous-tend le système de recherche d'information Wais), une séparation est faite entre ce que l'on a droit d'interroger et ce que l'on a droit de récupérer. À titre d'exemple, il est possible de spécifier que les champs nom et date de naissance peuvent être exportés, tout en n'autorisant que les requêtes sur le champ texte. Capturer ce type de capacité opérationnelle se fait très simplement en YAT en rajoutant un prédicat pour chaque attribut que l'on peut interroger mais pas retourner en résultat.

De nouvelles opportunités d'optimisation. Dans le but de bénéficier du potentiel d'optimisation fourni par de nouvelles opérations, l'optimiseur du système YAT doit être informé des relations entre ces opérations et celles qui sont déjà présentes dans le médiateur. En d'autres termes, nous devons pouvoir rajouter de nouvelles équivalences de la même façon que les optimiseurs de requêtes extensibles [Loh88]. Supposons que l'on soit intéressé par l'utilisation du prédicat contains de XML pour optimiser des sélections sur les champs de nos documents XML. Nous devons alors seulement expliquer la relation qui existe entre une telle sélection (impliquant le prédicat d'égalité) et contains. Ceci peut être fait grâce à l'équivalence suivante, qui stipule que l'on peut ajouter un prédicat plein-texte contains à tout prédicat portant sur un mot (String) d'une biographie, sans modifier la sémantique.

`biographies :` $\quad bios \xrightarrow{\star} \boldsymbol{Yat(X)}.[\boldsymbol{Bio}{:}\_],$	$\equiv$	`biographies :` $\quad bios \xrightarrow{\star} \boldsymbol{Yat(X)}.[\boldsymbol{Bio}{:}\_],$
$X = Y : String$		$X = Y : String,$ `contains`$(Y, \boldsymbol{Bio})$

6.3.3 Instanciation d'opérations

En YAT, les informations sur les structures de données des sources sont obtenues en raffinant le motif le plus général ($\boldsymbol{Yat}$) afin de décrire la source le plus précisément possible. De façon très naturelle, le mécanisme d'instanciation peut être utilisé pour raffiner les signatures des fonctions du modèle opérationnel. Cela réduit leur champ d'application sans pour autant changer leur sémantique (d'une façon similaire au polymorphisme paramétrique). Il faut alors remarquer que certains de nos opérateurs les plus importants (*Bind* et *Tree*) utilisent de filtres en entrée. Dans un premier temps, il nous faut donc être capable de typer ces motifs, c'est à dire de décrire les familles de motifs que l'on peut appliquer. Ceci nécessite une extension du mécanisme d'instanciation.

Décrire les capacités de filtrage et de liaison de variable des sources. Le filtrage et la liaison des variables sont deux mécanismes essentiels dans les langages de requêtes (cf. les variables liées dans la clause `from` d'OQL, les filtres de sélection dans la clause `where`, etc). Cependant, la plupart des langages de requête ne fournissent que des capacités réduites en comparaison des filtres YAT. Par exemple, il est rare qu'ils autorisent l'interrogation d'informations relatives au schéma. Dans le but de capturer ces limitations, nous étendons le modèle de données avec de nouvelles entités, appelées F-Motifs, dont la fonction est de décrire l'ensemble des filtres acceptés par une source donnée. Les F-Motifs sont, une fois de plus des motifs, auxquels nous rajoutons des *marqueurs*. Ces marqueurs sont de deux sortes. Les marqueurs **c** (pour « concret ») imposent que cette partie du filtre à laquelle ils sont appliqués soit concrète, c'est à dire complètement instanciée. Par là même, ils interdisent l'utilisation de variables. Inversement, les marqueurs **n** (pour « non instancié») interdisent toute instanciation.

La figure 6.9 décrit les F-Motifs correspondant à nos deux sources O_2 et XML-Wais. Chaque nœud ou arête peut être annoté par un ou deux marqueurs **c** ou **n** correspondant aux emplacements des variables des filtres. De même que pour les annotations de variables, à

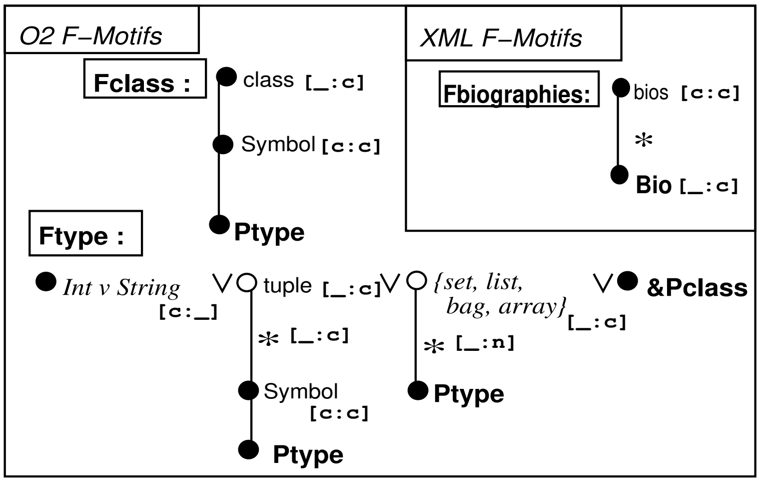


FIG. 6.9: *F-Motifs pour les sources O_2 et XML-Wais*

gauche ils sont appliqués sur les variables de motifs, à droite ils sont appliqués sur les variables d'annotation ou d'arité.

Les marqueurs sur les F-Motifs O_2 sont utilisés pour prévenir l'utilisation de variables sur les nœuds contenant des informations du schéma ou des méta-informations (p.ex. nom de la classe, noms d'attributs). De plus, ils assurent que l'on ne liera pas de variable de motif à un endroit du motif qui ne correspond pas à une valeur O_2 (par exemple au milieu d'un n-uplet). Le marqueur **n** sur l'arête correspondant aux collections est utilisé pour interdire le filtrage de collections avec une arité précise : dans OQL, on ne peut qu'itérer dans une collection !

Les marqueurs des F-Motifs XML sont bien plus simples : en fait ils interdisent tout, sauf la liaison d'une variable de motif sur la racine d'une biographie.

Les F-Motifs servent à décrire des familles de filtres, et ainsi fonctionnent comme des « types » pour les filtres YAT. Ces types peuvent être utilisés dans la signature d'opérateurs *Bind* instanciés par les sources, décrivant par là même quels sont leurs capacités en termes de filtrage et de liaison de variables.

Nous illustrons maintenant comment l'instanciation des signatures de nos opérateurs algébriques peut être réalisée pour décrire les capacités opérationnelles de notre source O_2 (i.e. OQL sur les motifs ODMG) ainsi que les capacités plein-texte de notre source XML.

Exemples pour OQL et XML-Wais. Le système O_2 est livré avec un interprète pour OQL [Cat97]. OQL peut réaliser la quasi totalité des opérations de YAT, mais avec cependant quelques restrictions. Ces restrictions concernent : (i) le type des données qui peuvent être interrogées qui doivent avoir une structure ODMG, et (ii) le fait qu'il est impossible en YAT_L d'interroger le schéma. Cette information peut être fournie au médiateur en spécialisant la signature de l'opérateur *Bind* d'origine. sur la figure 6.8, on peut voir que l'opérateur *Bind* a été spécialisé pour O_2. On peut remarquer que la signature du *Bind* O_2 contient le motif O_2 qui indique que l'on ne peut appliquer cette opération que sur des données ayant une structure ODMG et le F-Motif O_2 plutôt que le F-Motif YAT plus général. Remarquons aussi que l'adaptateur O_2 a été ajouté comme supportant cette opération.

De façon similaire, l'intégration des capacités de la source XML-Wais est réalisée en spécialisant l'opérateur *Bind* avec les motifs et F-Motifs XML appropriés. Notez que l'opération de sélection peut être évaluée par les deux systèmes intégrés sans que cela ait donné lieu à une spécialisation. Celle-ci est rendue inutile par le fait qu'une opération ne peut être poussée que si ses paramètres l'ont été. Par exemple, une sélection ne sera poussée sur O_2 que si (i) son prédicat est poussable et (ii) un *Bind* a été poussé précédemment, ce qui garantit un bon typage de l'opération.

6.3.4 Pousser l'évaluation vers les sources

Revenons maintenant à notre scénario d'intégration. Nous expliquons maintenant comment le médiateur décide si il peut pousser une opération vers une source ou pas. Partant d'une expression algébrique (p.ex. la règle d'intégration YAT$_L$ traduite dans l'algèbre donnée sur la figure 6.7), l'optimiseur effectue un parcours du bas de l'expression vers sa racine. À chaque étape, il détermine si il est possible de pousser l'évaluation vers la source interrogée. Dans l'exemple, les premières opérations sont les racines de persistance `artistes` et `biographies` qui peuvent toutes les deux être poussées, respectivement vers O_2 et XML-Wais. Du côté d'O_2, l'opération suivante est l'opération *Bind*. Il faut donc déterminer si celui-ci peut être ou non évalué par O_2. Cela se fait en confrontant le filtre utilisé avec le F-Motif O_2, qui, dans le cas présent, est bien compatible. En remontant ainsi, l'optimiseur sait qu'il doit s'arêter après le *DJoin* : l'opération de jointure qui suit dépendant de deux sources, elle n'est évaluable par aucune. Cette séquence d'opérations sera donc envoyée à l'adaptateur d'O_2 qui pourra le transformer en une requête OQL native comme celle-ci :

```
select  struct(N: x.nom, S: x.mouvement, A: y, Na: y.nom, Da: y.date)
from    artistes x, x.oeuvre y
```

Du côte XML, il n'est rien possible de faire car : (i) l'opération de *Bind* est trop complexe pour être poussée, et aucun prédicat de sélection plus haut ne peut être poussé vers la source.

6.4 Optimisation dans différents contextes d'intégration

Dans cette section, nous illustrons le potentiel d'optimisation fourni par YAT grâce aux différents mécanismes que nous venons de mettre en place. Nous imaginons qu'un utilisateur envoie des requêtes sur la vue intégrée définie dans la section 6.1. Nous choisissons d'utiliser YAT$_L$ comme langage d'interrogation pour plus de simplicité. Cependant, insistons sur le fait que les techniques d'optimisation proposées ici s'appliquent à tout langage qui peut être traduit dans l'algèbre (p.ex. SQL, OQL ou encore Lorel ou StruQL sans expressions de chemin généralisées).

En particulier, les exemples ont été choisis pour démontrer les capacités suivantes :

- Grâce à sa compréhension précise des sources, le médiateur YAT a la possibilité d'optimiser des requêtes structurées en utilisant des prédicats textuels.
- Il est possible d'interroger et même dans certains cas d'optimiser l'accès à des structures de données incomplètes (champs additionnels).
- L'optimiseur peut servir de la même manière à optimiser des requêtes sur la vue ou des accès par navigation.

6.4.1 Optimisation de requêtes avec des prédicats plein-texte

Notre premier exemple tire parti de la présence du prédicat `description_contains` que nous avons ajouté à la vue. Rappelons que cette opération a été définie par rapport au prédicat d'indexation plein-texte `text_contains`. Considérons maintenant la requête suivante :

Q1: *Quel est le nom des artistes impressionnistes qui ont participé à une exposition en 1874 ?*

$$
\mathit{noms} \xrightarrow{\star} N \quad \Longleftarrow \quad
\begin{array}{l}
\texttt{tous\_les\_artistes :} \\
\quad \mathit{root} \xrightarrow{\star} \boldsymbol{PA} : \\
\quad\quad \mathit{artiste}\langle| \rightarrow \mathit{nom} \rightarrow N, \\
\quad\quad\quad\quad\quad \rightarrow \mathit{mouvement} \rightarrow M \;|\rangle, \\
M = \text{“Impressionniste”}, \\
\texttt{description\_contains}(\text{“exposition de 1874”}, \boldsymbol{PA})
\end{array}
$$

Cette requête extrait l'identifiant $\boldsymbol{PA}$, ainsi que les nom (N) et mouvement (M) des artistes de la racine de persistance `tous_les_artistes`. Elle sélectionne ensuite ceux d'entre eux dont le mouvement est égal à “Impressionniste” et dont la description contient “exposition de 1874”. Enfin, elle construit comme résultat, un motif ayant un fils pour chaque nom distinct. On peut remarquer l'utilisation, naturelle dans ce contexte d'interrogation, du filtrage partiel (noté $\langle| \ldots |\rangle$, cf. section 5.4).

En suivant le processus décrit dans la section 6.2.3, la requête est tout d'abord convertie dans l'algèbre. Elle est ensuite combinée avec l'expression algébrique correspondant à la définition de la vue. La figure 6.10 illustre ceci. On peut remarquer que la limite entre l'expression algébrique de la requête et celle de la règle qui définit la vue, se caractérise par une séquence d'opérations $Bind - Tree$. La figure 6.11 donne l'expression après optimisation : cela montre à quel point l'expression peut être simplifiée. Avant d'expliquer comment cette expression optimisée a été obtenue, regardons tout d'abord ce qu'elle fait.

Tout d'abord, la source XML est interrogée pour retourner les biographies dont le champ texte contient la chaîne de caractères “exposition de 1874”. Pour évaluer cette requête la source pourra tirer parti de son index sur les champs texte. Ensuite, un deuxième $Bind$ est appliqué, au niveau du médiateur cette fois pour extraire les informations utiles dans les biographies (ici le nom de l'artiste). Puis, pour chacun de ces noms, la source O_2 est appelée et retourne le mouvement artistique de la personne. Ceci est dû à l'utilisation du DJoin sur la variable N, qui permet d'éviter ainsi de récupérer l'ensemble des artistes impressionnistes et d'effectuer la jointure localement dans le médiateur. Pour finir, l'opération $Tree$ construit le résultat. Voyons maintenant comment cette expression est trouvée. L'optimisation repose sur une stratégie linéaire en trois phases.

La 1$^{\text{re}}$ phase consiste à simplifier l'expression algébrique autant que possible. En particulier la séquence $Bind - Tree$ est réécrite, (ii) les sélections et les projections sont poussées, et (iii) les jointures inutiles sont éliminées :

– Comme on peut le voir sur la figure 6.10, la séquence $Bind - Tree$ se réécrit en une projection avec renommage. Cette réécriture se base sur la relation d'instanciation

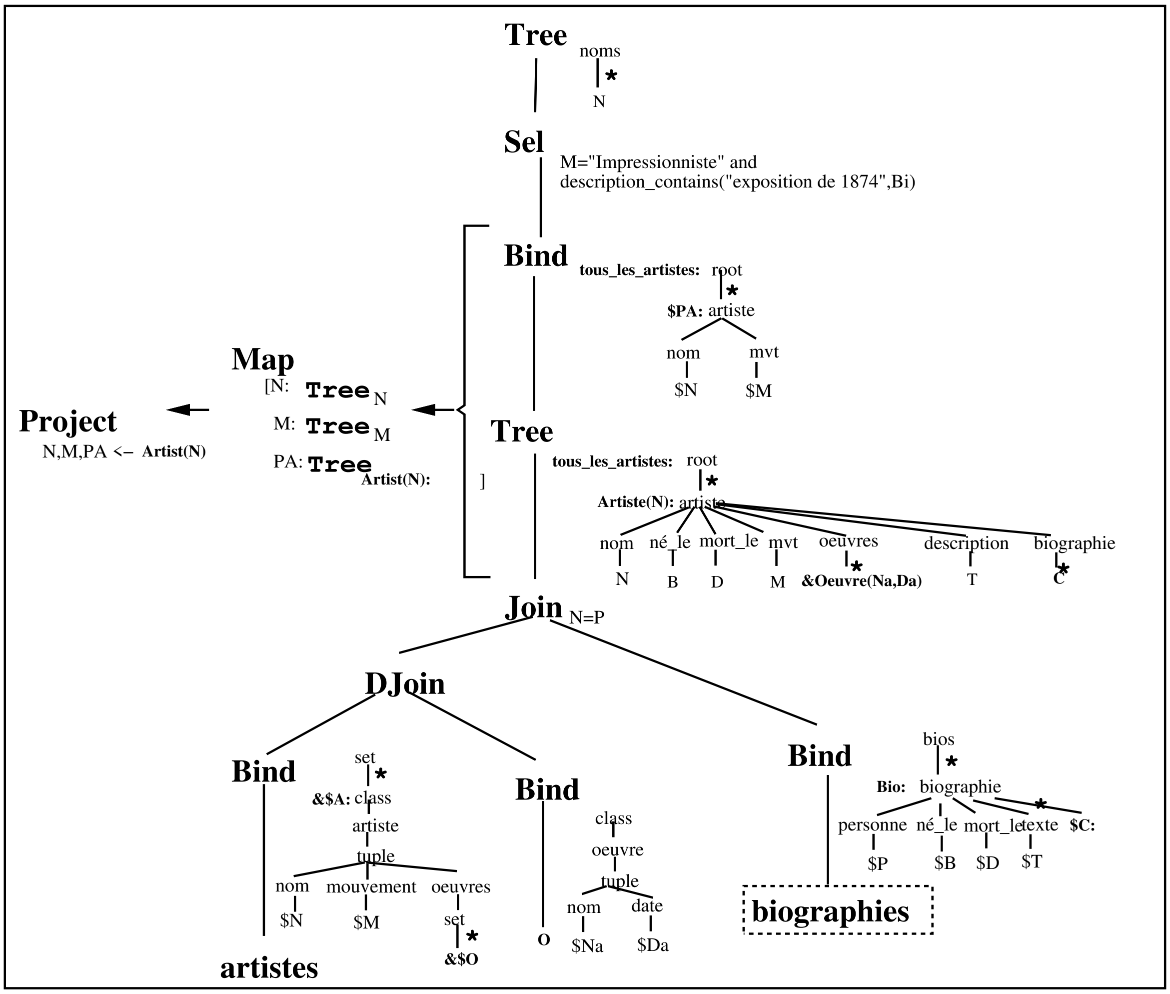


FIG. 6.10: *Algébraisation de Q1*

existant entre les filtres du *Bind* et du *Tree*. Les variables N et S sont mises en correspondance dans les deux filtres, tandis que la variable $\boldsymbol{PA}$ conserve le résultat de la fonction de Skolem $\boldsymbol{Artiste}(N)$. Notons qu'il est indispensable de conserver l'identifiant du motif correspondant à l'artiste pour pouvoir appliquer la fonction description_contains.

– La sélection utilise deux prédicats. Le premier (S = "Impressionniste") peut être poussé jusqu'au *Bind* portant sur les artistes. La seconde sélection est plus délicate à exploiter car elle implique la fonction exportée sur la vue. Il est donc nécessaire d'utiliser la définition de cette fonction (voir section 6.1) pour aller plus loin. Remarquons que cette réécriture de la fonction description_contains en la fonction text_contains nécessite de rajouter une liaison (Bi) pour l'identifiant des biographies XML.

– L'utilisation d'un regroupement sur N dans le filtre de sortie de la requête permet de choisir une sémantique ensembliste pour l'ensemble des opérateurs algébriques. Cela permet de pousser la projection à travers la jointure. On ne peut cependant

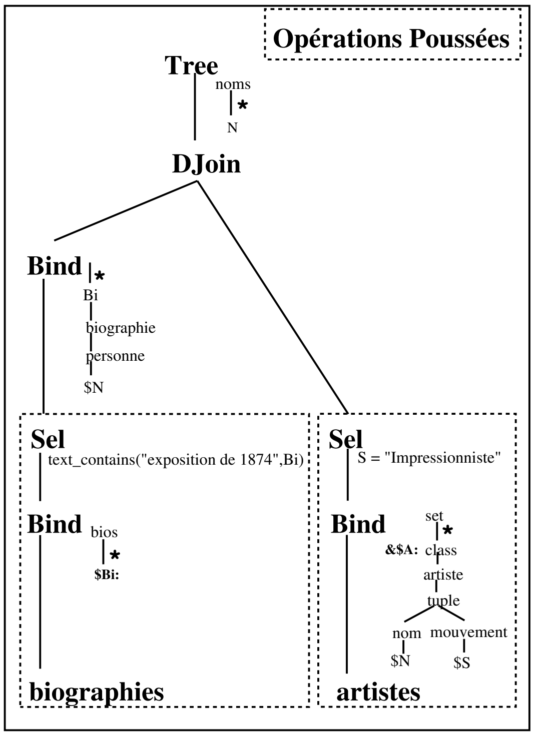


FIG. 6.11: *Optimisation de Q1*

faire mieux sans connaissance sémantique supplémentaire. Pour se débarrasser du *DJoin*, il faut que tous les artistes aient au moins une œuvre. Si c'est le cas, la jointure peut alors être supprimée.

La 2e phase est effectuée à partir du bas de l'expression algébrique et consiste à déterminer quelles parties de l'expression peuvent être poussées vers les sources. Ceci a déjà été expliqué dans la section précédente. Pour la source XML-Wais, nous souhaiterions pouvoir évaluer le prédicat de sélection dans la source. Cela peut être réalisé en effectuant une coupure dans le *Bind*, puis en poussant la sélection avec le premier *Bind* vers XML-Wais.

La 3e phase considère des d'optimisations plus avancées, en particulier la possibilité de transformer les *Join* en *DJoin* pour tirer parti de l'évaluation distribuée en passant des informations d'une source à l'autre. Il s'agit là d'une technique classique des bases de données distribuées [OV91].

6.4.2 Interrogation de données peu structurées

Nous illustrons maintenant comment il est possible pour l'utilisateur d'interroger les champs additionnels issus de la source XML, et pour nous d'optimiser autant que possible l'accès à ces informations. Dans la requête suivante, nous imaginons que l'utilisateur souhaite obtenir des informations tirées de certaines biographies proposant le champ additionnel endroit_favori.

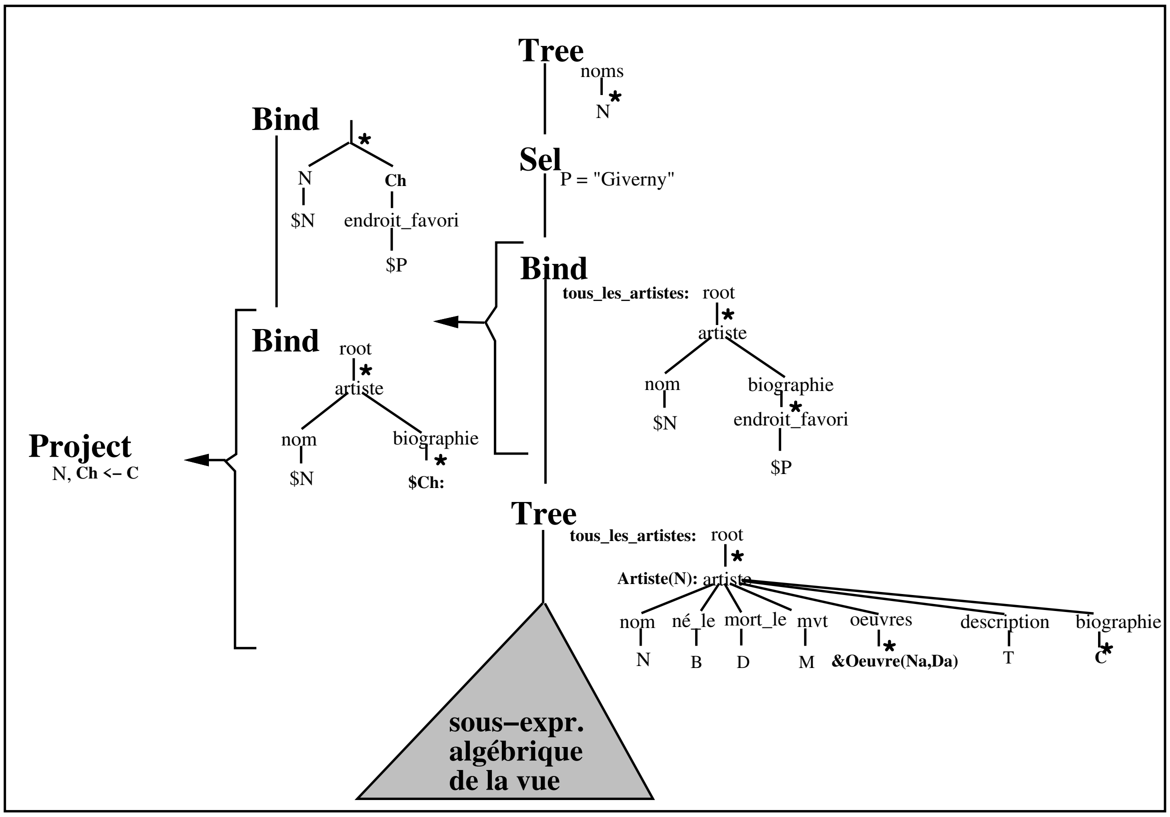


FIG. 6.12: *Algébraisation de Q2*

Q2: *Quels sont les noms des artistes qui ont apprécié "Giverny"?*

$$noms \xrightarrow{\star} N \quad \Longleftarrow \quad \begin{array}{l} \texttt{allartists:} \\ \quad root \xrightarrow{\star} artiste\langle| \rightarrow nom \rightarrow N, \\ \qquad\qquad\qquad\quad \rightarrow biographie \rightarrow endroit\_favori \rightarrow P \;|\rangle, \\ P = \text{"Giverny"} \end{array}$$

Cette requête retourne les noms des artistes dont le motif contient le champ additionnel endroit_favori, celui-ci ayant la valeur "Giverny". La figure 6.12 montre la traduction algébrique de cette requête et le début de l'optimisation.

Notons que dans l'expression originale, l'équivalence $Bind - Tree$ utilisée dans la requête précédente ne peut être appliquée directement, car les motifs ne sont pas instances l'un de l'autre. Pour cette raison, la simplification s'opère en deux temps. Tout d'abord, une coupure du *Bind* est réalisée au niveau du motif ***Champ***, l'équivalence $Bind - Tree$ peut ensuite être utilisée. Une fois ceci fait, la simplification se passe de la même façon que pour la requête précédente : on pousse la sélection et la projection puis on élimine les jointures inutiles, ce qui donne finalement l'expression à gauche de la figure 6.12.

Maintenant, si l'on évalue la requête telle qu'elle, on peut remarquer que l'on n'exploite pas les capacités de requête sur le contenu offert par Wais. De fait, on souhaiterait ne pas

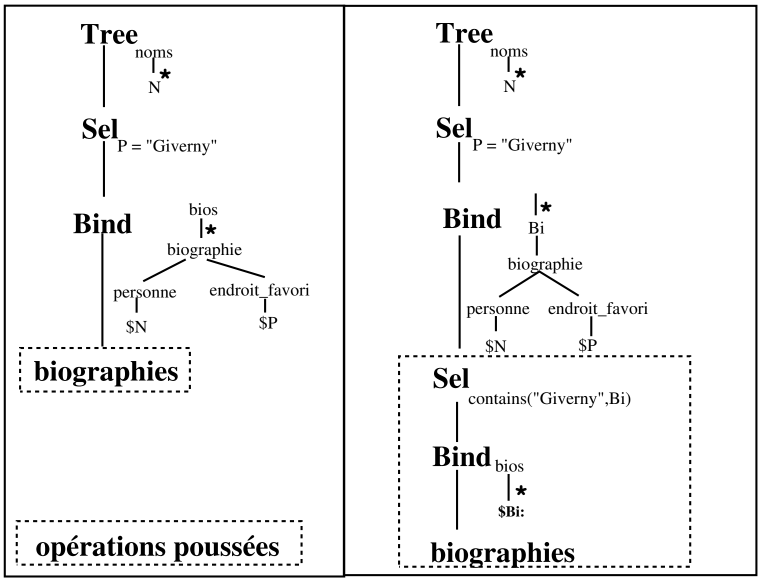


FIG. 6.13: *Optimisation de Q2*

récupérer toute la collection de biographies mais seulement celles qui contiennent "Giverny", procédant ainsi à une pré-sélection. Ceci peut être réalisé grâce à l'équivalence que nous avons introduit dans la section 6.3.2. Rappelons nous que cette équivalence autorise l'ajout d'un prédicat de sélection impliquant l'index plein-texte avant certaines sélections structurées (comme c'est le cas sur la requête). Nous pouvons donc : (i) rajouter cette sélection, (ii) couper le *Bind* en deux au niveau de la racine des biographies, et (iii) pousser le premier *Bind* ainsi que la sélection vers la source XML. Le résultat de cette réécriture est donné à droite de la figure 6.13.

6.4.3 Optimiser les accès par navigation

L'une des caractéristiques intéressantes de YAT$_L$ est qu'il permet la définition de références. L'utilisateur peut indiquer la granularité avec laquelle les références apparaissent. Il s'agit d'une propriété intéressante qui donne à l'utilisateur la possibilité de naviguer dans un graphe de données YAT à la manière d'un hypertexte.

Le cas de la navigation est particulièrement intéressant car simplifiée par l'usage de fonctions de Skolem. On peut, en effet, voir les fonctions de Skolem comme une structure d'index particulière, utilisée pour les identificateurs. De fait, la navigation se fait par accès direct à un objet et se traduit dans l'algèbre comme une sélection sur les valeurs des paramètres de la fonction de Skolem (qui sont souvent des clés, donc très sélectifs).

Expliquons maintenant comment les accès par navigation sont évalués. C'està-dire, que se passe-t'il lorsque l'utilisateur « clique » sur un identifiant en souhaitant récupérer la valeur correspondante ? Ceci correspond à l'appel d'une fonction de Skolem avec des paramètre déterminés. Par exemple, imaginons qu'un utilisateur accès à la racine de persistance `tous_les_artistes`, puis souhaite accéder à l'identifiant correspondant à l'une des œuvres de Cézanne (p.ex. celui correspondant aux "Baigneurs", 1891). Cet appel est converti dans une requête vers le médiateur. L'identifiant demandé est mis en correspondance avec l'appel à la fonction de Skolem ***Oeuvre***("Baigneur",1891) qui doit être calculé.

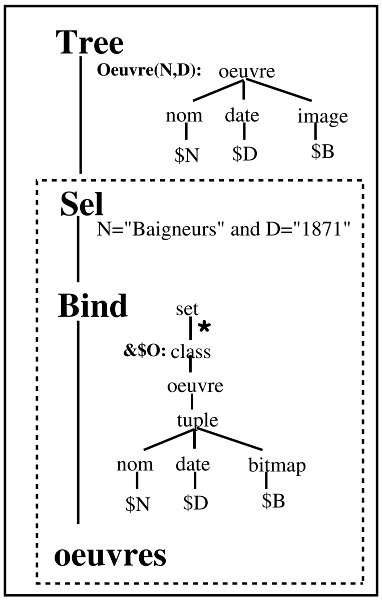


FIG. 6.14: *Optimisation des accès par navigation*

L'évaluation correspond donc à une sélection sur les variables N et D (respectivement égales à "Baigneur" et 1891). Il s'agit donc d'une requête que l'on peut facilement optimiser en poussant les sélections. Le résultat de l'optimisation est montré sur la figure 6.14.

6.5 Une formalisation : algèbre et équivalences

Cette section propose une définition formelle des opérateurs de l'algèbre ainsi que la démonstration des équivalences qui ont été utilisées tout au long de ce chapitre.

6.5.1 Transposition d'une algèbre objet sur les structures Tab

Comme nous l'avons déjà précisé, la quasi totalité des opérateurs de notre modèle opérationnel sont ceux de l'algèbre objet de Cluet et Moerkotte [CM93]. Nous indiquons ici brièvement leurs définitions, transposées aux structures de Tab. Notons qu'il s'agit d'une algèbre *multi-sortes*, c'est à dire que tous les opérateurs sont définis sur trois types de collections de n-uplets (ensembles, multi-ensembles et listes). Pour ne pas alourdir la présentation, nous ne donnerons les définitions que sur les ensembles. Celles-ci s'étendent trivialement aux deux autres types de collections.

Structures de Tab. Dans la section 4.2.1, nous avions introduit les structures de Tab pour représenter les collections de variables liées au cours du filtrage. Les structures de Tab remplacent les collections de n-uplets utilisées habituellement en bases de données dans les langages algébriques. De la sorte, ils permettent de rester dans un modèle de données homogène. Pour des raisons de lisibilité, nous revenons maintenant à une notation utilisant des collections de n-uplets. Bien que cela puisse paraître surprenant, insistons sur le fait qu'il ne s'agit là que d'une notation pour les structures de Tab. Le modèle de données reste donc le même.

Ainsi, la notation :

$$\{[A_1 : a_{11}; A_2 : a_{12}; \ldots; A_n : a_{1n}], \ldots, [A_1 : a_{k1}; A_2 : a_{k2}; \ldots; A_n : a_{kn}]\}$$

correspond à la structure de Tab suivante :

$$\begin{array}{lll}
_\{ \rightarrow & _\{ & \rightarrow A_1 \rightarrow a_{11}, \\
& & \rightarrow A_2 \rightarrow a_{12}, \\
& & \ldots, \\
& & \rightarrow A_n \rightarrow a_{1n}\} \\
\ldots, & & \\
\rightarrow & _\{ & \rightarrow A_1 \rightarrow a_{k1}, \\
& & \rightarrow A_2 \rightarrow a_{k2}, \\
& & \ldots, \\
& & \rightarrow A_n \rightarrow a_{kn}\}\}
\end{array}$$

Opérations élémentaires sur les structures de Tab.

- **concaténation :** L'opération $\circ$ dénote la fonction de concaténation sur les "n-uplets" (*i.e.* sur les éléments d'un Tab).
- **navigation :** L'opération $t.A$ dénote la fonction d'extraction de la valeur de la variable A dans le n-uplet t.
- **élimination :** L'opération $t.\vec{A}$ retourne la sous-partie du n-uplet t constituée des attributs $\vec{A}$.

Opérateurs algébriques. L'algèbre propose les opérations ensemblistes traditionnelles, telles que l'union ($\cup$), l'intersection ($\cap$), la différence ($\setminus$) et le produit cartésien ($\times$), ainsi que les opérations suivantes.

- **Map :** L'opérateur Map applique une même fonction f sur une collection. Il est donc défini comme suit :

  $$Map_f(e) = \{f(t) | t \in e\}$$

  La figure 6.6 donne un exemple d'application du Map. Les trois cas particuliers suivants sont également utilisés en pratique :

  1. $e[A] = \{[A : x] | x \in e\}$ construit un nouveau Tab ayant un fils pour chaque donnée en entrée.
  2. $Map_{\circ[A:f]}(e) = \{t \circ [A : f(t)] | t \in e\}$ ne fonctionne que sur des expressions e de type Tab. Elle évalue la fonction f sur chaque fils, puis concatène à celui-ci une liaison entre la variable a et la valeur résultante. Nous utiliserons $Map_{A:f}(e)$ pour dénoter $Map_{\circ[A:f]}(e)$.
  3. $Map_{\circ f}(e) = \{t \circ f(t) | t \in e\}$ évalue la fonction f sur chaque fils, puis concatène le résultat à celui-ci.

- **Jointure (Join) :** $e_1 \bowtie_p e_2 = \{t_1 \circ t_2 \mid t_1 \in e_1, t_2 \in e_2, p(t_1, t_2)\}$, où p est un prédicat booléen sur les couples de n-uplets. Cette définition suppose que les variables de t_1 et t_2 sont disjointes.

- **Jointure dépendante (DJoin) :** $(e_1 < e_2 >) = \{y \circ x | y \in e_1, x \in e_2(y)\}$. La jointure dépendante est une jointure entre deux ensembles dont l'un dépend de l'autre. Ici, l'évaluation de e_2 est paramétrée par les éléments y, résultant de l'évaluation de e_1.

- **Sélection (Select) :** $\sigma_p(e) = \{t | t \in e, p(t)\}$ sélectionne les n-uplets d'une expression algébrique e qui satisfont un prédicat p. Un exemple de sélection est donné sur la figure 6.6.

- **Projection (Project) :** $\pi_{\vec{A}}(e) = \{t.\vec{A} \mid t \in e\}$. La projection sur un attribut A_i avec $A_i \notin \vec{A}$ n'est pas définie.

- **Group :** $Group_{G;\vec{A}_1;f}(e) = \{t_1.\vec{A}_1 \circ [G : g] | t_1 \in e, g = \{f(t_2) | t_2 \in e, t_2.\vec{A}_1 = t_1.\vec{A}_1\}\}$. Cette opération regroupe dans la variable G, l'ensemble des résultats de l'évaluation de f sur les n-uplets de e qui partagent une même valeur pour $\vec{A}_1$.

- **Tri :** $Sort_{\vec{A}}(e) = \langle t_1, t_2, ..., t_n \rangle$ tel que
$$\begin{aligned} &(\forall i \in [1, n], t_i \in e) \\ &\wedge(\forall t \in e, \exists i \in [1, n], t_i = t) \\ &\wedge(\forall i \in [1, n-1]\ t_i.\vec{A} \leq t_{i+1}.\vec{A}) \end{aligned}$$
Cette opération trie le résultat de e par ordre croissant sur les valeurs de $\vec{A}$. Elle produit toujours une liste. Le tri est non déterministe lorsque deux entrées ont la même valeur pour le critère de tri.

6.5.2 Opérateurs Bind et Tree

L'intuition qui sous-tend ces deux opérateurs a déjà été présentée dans la section 4.2.1. Quant à leur usage, il a été illustré tout au long de ce chapitre. Tous deux capturent les manipulations spécifiques au modèle semistructuré et se fondent sur la notion d'instanciation (cf. définition 4.3.7).

Bind sur les variables. Nous commençons par définir la liaison sur les symboles pouvant apparaître dans les annotations de variables.

Définition 6.5.1 *Bind de variables.* Soit V une variable dans $\mathbf{V}$. Soit v une valeur du modèle YAT (label ou valeur de motif), c'est à dire $v \in \mathbf{C} \cup \mathbf{N} \cup \mathbb{N}$, alors :

- $bind(V, v) = [V : v]$;

- $bind(V, \_) = []$.

□

Opérateur Bind. Le *Bind* se définit à partir de l'instanciation de la façon suivante. Dans le cas où un motif concret est instance du motif d'un filtre. On en déduit une structure de Tab qui conserve des liaisons pour les variables du filtre. La fonction rank retourne le rang du nœud fils dans un motif.

Définition 6.5.2 *Opérateur Bind.* Soit F une forêt complète, $f = t[V_1, \dots, V_n]$ un filtre de motif t et de variables $V_1, \dots, V_n$ dans $\mathbf{V} \cup \{\_\}$. soit t' un motif concret, instance de t dans F. Alors l'opération *Bind* de f sur t' est défini récursivement de la façon suivante :

1. si $f = l.[V_i : V_j]$ et $t' = l'$ alors $Bind_f(t') = \{bind(V_i, t') \circ bind(V_j, l')\}$;
2. si $f = \boldsymbol{\&p}.[\&V_i : V_j]$ et $t' = \boldsymbol{\&p'}$ alors $Bind_f(t') = \{bind(V_i, \boldsymbol{p'}) \circ bind(V_j, l')\}$;
3. si $f = \ \_\langle \overset{\epsilon.[V_i:V_j]}{\rightarrow} f_1\rangle$ et $t' = l'\langle \rightarrow t'_1, \dots, \rightarrow t'_k\rangle$, alors :
$$Bind_f(t') = \{bind(V_i, t')\} \times \bigcup_{i=1..n} (\{bind(V_j, \mathsf{rank}(t'_i))\} \times Bind_{f_1}(t'_i)) \ ;$$
4. si $f = l.[V_i : V_j]\langle \overset{\epsilon_1}{\rightarrow} t_1, \dots, \overset{\epsilon_n}{\rightarrow} t_n\rangle$, et $t' = l'\langle \rightarrow t'_{1,1}, \rightarrow t'_{1,2}, \dots, \rightarrow t'_{1,m_1}, \dots, \rightarrow t'_{n,m_n}\rangle$, tel que $\forall i \in [1..n] \quad l'\langle \rightarrow t'_{i,1}, \rightarrow t'_{i,2}, \dots, \rightarrow t'_{i,m_i}\rangle$ est une instance de $l\langle \overset{\epsilon_i}{\rightarrow} t_i\rangle$ dans F, alors :
$$\begin{aligned} Bind_f(t') = \{bind(V_i, t') \circ bind(V_j, l')\} \ &\times Bind_{\_\langle \overset{\epsilon_1}{\rightarrow} t_1\rangle}(l' \ \langle \rightarrow t'_{1,1}, \dots, \rightarrow t'_{1,m_i}\rangle) \\ &\times \dots \\ &\times Bind_{\_\langle \overset{\epsilon_n}{\rightarrow} t_n\rangle}(l' \ \langle \rightarrow t'_{n,1}, \dots, \rightarrow t'_{n,m_i}\rangle) \, ; \end{aligned}$$
5. si $f = L\{\overset{\epsilon_1}{\rightarrow} t_1, \dots, \overset{\epsilon_n}{\rightarrow} t_n\}$, et $t' = L'\{\rightarrow t'_1, \rightarrow t'_2, \dots, \rightarrow t'_m\}$, et il existe une permutation σ de $[1..m]$, telle que $t'' = L'\langle \rightarrow t'_{\sigma(1)}, \rightarrow t'_{\sigma(2)}, \dots, \rightarrow t'_{\sigma(m)}\rangle$ est une instance de $t = L\langle \overset{\epsilon_1}{\rightarrow} t_1, \dots, \overset{\epsilon_n}{\rightarrow} t_n\rangle$ dans F, alors :
$$Bind_f(t') = Bind_f(t'') \ ;$$
6. $Bind_f(t') = \{[]\}$ dans tous les autres cas.

□

Définition du Tree. L'opérateur *Tree* effectue l'opération inverse du *Bind*. Une manière intuitive de le définir serait donc : pour un filtre f et une structure *tab* d'entrée, comme le motif t', tel que : appliquer de nouveau un *Bind* sur t' avec le filtre f nous redonne bien *tab* comme résultat. Malheureusement, cela ne suffit pas. En effet, dans beaucoup de cas, plusieurs t' distincts vérifient cette propriété (cf. figure 4.11). C'est pourquoi les filtres de sortie, c'est à dire les filtres utilisés par l'opérateur *Tree*, contiennent des primitives de regroupement sur les arêtes. Celles-ci permettent de lever les ambiguités.

Nous définissons donc de nouveau l'opérateur *Tree* récursivement. Dans cette définition, la fonction *pick* dénote la fonction retournant l'un des éléments d'un ensemble. Cette fonction est la principale source de non déterminisme de l'opérateur *Tree*. Rappelons que nous

nous sommes restreints dans les exemples d'optimisation aux seules transformations déterministes. Les contraintes imposées dans la section 5.1.5 permettent d'assurer le déterminisme, ce qui se traduit au niveau de l'opérateur $Tree$ par des fonctions *pick* toujours appliquées sur des singletons. Notons également que les équivalences décrites dans la section suivante ne fonctionnent que sur des opérations $Tree$ déterministes.

Définition 6.5.3 *Opérateur Tree.* Soit tab une structure de Tab et $f = t[A_1, \dots, A_n]$ un filtre de sortie, de motif t et de variables $A_1, \dots, A_n$. Alors l'opération $Tree$, de f sur tab est définie récursivement de la façon suivante :

1. si $f = l$ alors $Tree_f(tab) = l$;
2. si $f = A$ alors $Tree_f(tab) = pick(tab).A$;
3. si $f = \&\boldsymbol{p}(A_{i_1}, \dots, A_{i_k})$ alors $Tree_f(tab) = \&\boldsymbol{p}(t'.A_{i_1}, \dots, t'.A_{i_k})$ avec $t' = pick(tab)$;
4. si $f = \_\langle \stackrel{g(\vec{A})o(\vec{A}')}{\longrightarrow} f'\rangle$, avec $\vec{A}' \subset \vec{A}$, alors :

$$Tree_f(tab) = \_Map_{\,Tree_{f'}[G]}(Sort_{\vec{A}'}(Group_{G;\vec{A};id}(tab)))\ ;$$

5. si $f = f_0\langle \stackrel{\epsilon_1}{\to} f_1, \dots, \stackrel{\epsilon_n}{\to} f_n\rangle$, alors :

$$Tree_f(tab) = l\langle \to t_{1,1}, \to t_{1,2}, \dots, \to t_{1,m_1}, \dots, \to t_{n,m_n}\rangle$$

   avec $l = Tree_{f_0}(tab)$ et $\forall i \in [1..n],\ Tree_{\_\langle \stackrel{\epsilon_i}{\to} f_i\rangle}(tab) = \_\langle \to t_{i,1}, \to t_{i,2}, \dots, \to t_{i,m_i}\rangle$;

6. si $f = f_0\{\stackrel{\epsilon_1}{\to} f_1, \dots, \stackrel{\epsilon_n}{\to} f_n\}$, alors :

$$Tree_f(tab) = l\{\to t_1, \to t_2, \dots, \to t_m\}$$

   tel que, $Tree_{f_0\langle \stackrel{\epsilon_1}{\to} f_1, \dots, \stackrel{\epsilon_n}{\to} f_n\rangle}(tab) = l\langle \to t_1, \to t_2, \dots, \to t_m\rangle$.

□

Nous avons défini le $Tree$ indépendamment du *Bind*. Il faut donc nous assurer que cette définition correspond à la sémantique intuitive que nous avions donnée précédemment : autrement dit, que lorsqu'on applique une opération de *Bind* sur le résultat, on retrouve bien le tab d'origine. En fait, cette propriété ne se trouve vérifiée que dans le cadre d'une sémantique ensembliste et pour des opérations de $Tree$ déterministes. Cette propriété s'exprime à l'aide du théorème suivant.

Théorème 6.5.1 Soit f_i un filtre d'entrée de motif t et de variables $\vec{A}$. Soit f_o un filtre de sortie correspondant (*i.e.* le même filtre, avec des opérations de regroupement sur les arêtes). Soit tab, une structure de Tab ensembliste. Si $Tree$ est déterministe par rapport à tab (c'est-à-dire, s'il est déterministe par rapport aux dépendances fonctionnelles vérifiées par le tab), alors :

$$Bind_{f_i}(Tree_{f_o}(tab)) = \pi_{\vec{A}}(tab)$$

□

6.5.3 Nouvelles équivalences algébriques

Intéressons nous maintenant aux équivalences particulières vérifiées par les nouveaux opérateurs *Bind* et *Tree*.

Équivalences élémentaires.

Couper un Bind.

Théorème 6.5.2 *Coupure horizontale d'un Bind en deux Bind (profondeur 1).*
Soit $f = f_0\{\stackrel{\epsilon_1}{\rightarrow} f_1, \ldots, \stackrel{\epsilon_i}{\rightarrow} f_i, \ldots \stackrel{\epsilon_n}{\rightarrow} f_n\}$ un filtre d'entrée de variables $\vec{A}$.

$$Bind_f \equiv Bind_{\_\stackrel{\star}{\rightarrow}\_\{\rightarrow A_1 \rightarrow \_[\_:A_1],\ldots,\rightarrow B \rightarrow f'\}}(Bind_{f_0\{\stackrel{\epsilon_1}{\rightarrow} f_1,\ldots,\stackrel{\epsilon_i}{\rightarrow}\_[B:\_],\ldots\stackrel{\epsilon_n}{\rightarrow} f_n\}})$$

□

Cette propriété se démontre à partir de la définition du *Bind*.

Théorème 6.5.3 *Coupure horizontale d'un Bind en deux Bind (profondeur quelconque).*
Soit f un filtre d'entrée de variables $\vec{A}$, et f' un sous filtre de f de variables $\vec{A'} \subset \vec{A}$.

$$Bind_f \equiv Bind_{\_\stackrel{\star}{\rightarrow}\_\{\rightarrow A_1 \rightarrow \_[\_:A_1],\ldots,\rightarrow B \rightarrow f'\}}(Bind_{f[f'|B]})$$

□

Cette propriété se démontre par récurrence à partir de la coupure à profondeur 1.

Simplifier une séquence Bind-Tree.

Théorème 6.5.4 *Simplification $Bind - Tree$ avec élimination de variables.*
Soit f_i un filtre d'entrée de variables $\vec{A}$. Soit f_o un filtre de sortie correspondant mais de variables $\vec{A'} \supset \vec{A}$. Si $Tree$ est déterministe, alors :

$$Bind_{f_i}(Tree_{f_o}) \equiv \pi_{\vec{A}}$$

□

Cette équivalence découle de la propriété 6.5.1. et de l'élimination de variables dans le *Bind*.

Chapitre 7

Le système YAT

Beware of bugs in the above code;
I have only proved it correct, not tried it.
– Donald E. KNUTH

LE SYSTÈME YAT EST UN SYSTÈME D'INTÉGRATION SEMISTRUCTURÉ. Son objectif est de fournir les outils permettant de réaliser le plus rapidement et le plus simplement possible l'intégration de sources de données quelconques, tout en offrant les performances requises par les applications devant manipuler de gros volumes de données. Un premier prototype a été développé. Il implante la majeure partie des outils présentés dans cette thèse, à l'exception notable des techniques d'optimisation introduites dans le chapitre précédent. Ce prototype a servi à la réalisation de plusieurs applications d'intégration et de conversion. Il a également été utilisé par nos partenaires industriels dans le cadre du projet européen OPAL [OPA95].

Dans la section 7.1, nous donnons un rapide aperçu du système et de ses composants principaux. Dans la section 7.2, nous présentons l'architecture détaillée, en insistant sur les problèmes liés à la réutilisation des composants développés. Nous donnons également un état d'avancement précis pour chacun des composants. Dans la section 7.3 nous illustrons comment le système s'utilise, d'abord sur le principe, puis sur quelques applications. Nous montrons notamment comment l'architecture fonctionnelle permet de se rapprocher de plus en plus du cas de figure idéal, dans lequel l'utilisateur n'a d'autre tâche que de décrire l'intégration sémantique.

7.1 Aperçu du système et architecture générale

Nous commençons par expliquer brièvement le contexte dans lequel le développement du système YAT s'est déroulé et donnons un bref historique (section 7.1.1). Nous présentons ensuite les principaux composants de notre architecture adaptateur/médiateur (section 7.1.2).

7.1.1 Historique de YAT

Initialement, le système YAT a été développé dans le cadre du projet européen OPAL [1]. Le projet OPAL a pour objectif de fournir une plate-forme logicielle permettant l'intégration des

1. *Integrated Information and Process Management In Manufacturing Engineering*, Projet Esprit IV N° 20377. `http://www.cordis.lu/esprit/src/20377.htm`

systèmes utilisés lors de la conception et de la fabrication de produits manufacturés. Notre tâche dans ce projet concernait l'élaboration des outils pour l'intégration des données (tâche 1.3, *OPAL repository*), ainsi que pour la génération automatique de pages HTML à partir des données intégrées (tâche 1.5, *Hypermedia Facilities*).

Le système YAT est utilisé pour la génération de l'interface HTML : autrement dit, il constitue le noyau des *Hypermedia Facilities*. Du point de vue du travail présenté ici, il s'agit d'une application de conversion. Le premier prototype du système YAT a été délivré aux autres membre du projet en janvier 1998. Il a été intégré au reste de l'architecture d'OPAL en mars 1998. Il est impliqué depuis dans la réalisation de trois applications pilotes par les sociétés Bosch, PSA et Fatronik. Une première démonstration des ces applications, incluant l'utilisation de YAT pour l'interface, a été faite en octobre 1998. Le projet OPAL s'achève en janvier 1999.

Depuis janvier 1998, le développement de YAT s'est poursuivi autour de deux axes. Nous avons tout d'abord testé ses possibilités sur plusieurs applications, dont certaines sont décrites dans la section 7.3. Cela nous a également permis de prendre conscience des limites du système lors de la manipulation de gros volumes de données. Dans le même temps, le système a subit un réécriture complète, avec pour objectif de se diriger vers l'architecture décrite dans la section 7.2 et de préparer l'ajout de nouvelles fonctionnalités (principalement celles relatives à l'intégration d'opérations, cf. chapitre 6). Cette refonte est terminée depuis quelques semaines et l'implantation des nouveaux modules est maintenant en cours.

7.1.2 Architecture : rappel et vue d'ensemble

L'architecture sommaire du système a déjà été présentée au début de cette thèse (voir la figure 2.7). Il s'agit d'une architecture fondée sur deux composants principaux (adaptateur et médiateur) et des composants applicatifs. Cette architecture a été conçue pour répondre aux besoins essentiels dans un cadre d'intégration. De ce point de vue, YAT est conforme à l'architecture de référence préconisée par l'armée américaine [AHK95]. C'est à dire qu'elle permet d'obtenir la composabilité des composant, une réutilisation maximale des modules existants et qu'elle favorise un accès uniforme à toutes les sources. Nous regardons maintenant plus en détail chacun des composants, ainsi que leurs fonctions respectives.

Une interface uniforme. Les adaptateurs et médiateurs possèdent une interface d'accès commune qui permet de voir la source d'information sous-jacente (source originelle dans le cas d'un adaptateur ou intégrée dans le cas d'un médiateur) de façon standard et homogène. Plus exactement, cette interface fournit, pour une source de données quelconque, les services que nous avons distingués comme étant toujours disponibles (cf. section 2.2). Elle permet donc l'accès : (i) aux données elles-mêmes, (ii) aux informations sémantiques relatives à ces données (c.-à-d., sous les hypothèses que nous nous sommes fixées, à leur structure) et (iii) aux opérations susceptibles d'être appliquées par la source.

Adaptateurs d'entrée. Ils doivent exporter les différentes informations de la source dans le modèle pivot, en accord avec l'interface ci-dessus. L'accès à la source doit pouvoir se faire quels que soient le protocole et l'interface qu'elle supporte. Les adaptateurs transforment les opérations qui lui sont soumises en une requête compréhensible par la source, puis convertissent le résultat vers le modèle pivot.

Médiateurs. Le rôle du médiateur est de procéder à l'intégration, telle qu'elle est spécifiée en YAT_{L}. Il doit être capable d'évaluer (et d'optimiser !) les opérations soumises à la vue intégrée, quelle que soit la politique d'intégration – virtuelle ou materialisée – choisie.

Composants applicatifs. Un ensemble de composants, pouvant être utilisés pour les besoins d'applications spécifiques, est proposé en standard. Parmi ces composants, on trouve notamment des adaptateurs de sortie, permettant d'exporter les données intégrées vers un format et un système cibles. Ceux-ci sont utilisés par les applications de conversion. Le système fournit également des interfaces génériques pour la navigation et l'interrogation.

De fait, les adaptateurs d'entrée résolvent les problèmes d'hétérogénéité dont le traitement peut être automatisé (*i.e.* interopérabilité, hétérogénéité des opération et des formats), tandis que le médiateur se concentre uniquement sur l'intégration structurelle. Ce choix permet de bâtir des adaptateurs génériques, tout en réduisant au strict minimum le travail nécessaire pour décrire l'intégration de sources hétérogènes. Outre les fonctionnalités que nous venons d'évoquer, chaque composant fournit des services utiles lors de la phase d'intégration. L'utilisateur du système peut, par exemple, prendre connaissance des informations structurelles exportées par les différents composants, vérifier les programmes d'intégration qu'il écrit, etc.

7.2 Architecture détaillée

Avant de rentrer dans les détails de l'architecture, nous revenons sur quelques choix techniques importants (section 7.2.1) concernant l'implantation du système. Nous passons ensuite en revue chaque composant et donnons leur architecture détaillée. L'interface d'accès est décrite dans la section 7.2.2. Nous expliquons ensuite l'architecture interne des adaptateurs (section 7.2.3), des médiateurs (section 7.2.4) et de quelques composants applicatifs (section 7.2.5). L'architecture qui est décrite ici, inclut les modules nécessaires au traitement du modèle opérationnel. Pour distinguer clairement cette architecture « idéale » de celle du prototype actuellement disponible, nous donnons à la fin de chaque section un état d'avancement précis.

7.2.1 Quelques choix techniques

Langage d'implantation. Le système YAT est écrit en Objective Caml[2] [Ler98], un langage fonctionnel de la famille de ML développé à l'Institut National de Recherche en Informatique et en Automatique.

Nous avons choisi ce langage pour l'excellent compromis entre l'environnement de développement qu'il procure et les performances. Le langage est sûr, il permet de manipuler facilement des structures de données complexes. Il offre une librairie très complète ainsi qu'un bonne interface avec C. Il propose un compilateur natif qui produit un code exécutable extrèmement performant.

Un autre avantage de l'utilisation de Caml est la portabilité du code. Ainsi, le système YAT a déjà été compilé et testé sans aucun problème majeur sur les plate-formes suivantes : Sparc-SunOS, Sparc-Solaris, Intel-Linux et Intel-Windows95.

2. `http://caml.inria.fr/`, `http://caml.inria.fr/ocaml/`

Fonctions de Skolem. L'implantation des fonctions de Skolem nécessite un soin particulier. Elles sont implantées en utilisant les tables de hachages fournies avec les librairies Caml.

Fonctions externes. Les fonctions externes sont implantées dès le départ dans un module Caml bien précis. L'isolation du module et son organisation permettent de facilement rajouter une nouvelle fonction externe. Cependant, ce module doit être intégré dans le médiateur au moment de la compilation. Une implantation fondée sur l'utilisation du chargement dynamique de modules Caml est envisagée. Cela permettrait de réaliser plus simplement un module contenant les fonctions externes dont une application donnée a besoin et de l'utiliser sans recompiler le médiateur YAT.

Interface C Le reste du système OPAL est écrit en C++. Une interface C a donc été mise en place pour permettre l'accès aux services du système YAT à travers C++. Il s'agit d'une interface offrant des services minimums, plus réduits que ceux de la YAT-API.

7.2.2 Interface d'accès standard : YAT-API

Un aspect important de l'architecture fonctionnelle est l'existence d'une interface uniforme pour l'accès à tous les composants. Ainsi, il n'est pas possible de différencier adaptateurs et médiateurs, ce qui assure leur composabilité. On peut par exemple enchaîner plusieurs médiateurs, construisant de la sorte plusieurs niveaux d'intégration. Les principales primitives offertes par l'API du système sont les suivantes :

Initialisation. La première étape consiste à installer les différents composants : les adaptateurs doivent se « connecter » à une source, tandis que le médiateur doit se connecter à plusieurs autres composants YAT. Lors de cette phase préliminaire, une primitive d'initialisation permet de communiquer les paramètres indispensables. Dans le cas d'un adaptateur, ces paramètres sont spécifiques à la source : paramètres réseau, nom du répertoire contenant les documents à manipuler, nom de la base de données, URL du serveur à interroger sur la Toile, etc. Dans le cas d'un médiateur, il s'agit des noms de machines et numéros de port des autres composants YAT impliqués dans l'intégration. Une fois la connection réalisée, tous les autres services sont opérationnels.

Accès à la structure. Chaque source fournit une ou plusieurs descriptions de son contenu dans le modèle de données défini au chapitre 4. Les forêts YAT exportées peuvent être plus ou moins précises/génériques, permettant de choisir le niveau de représentation le mieux adapté à l'application. Par exemple, l'adaptateur O_2 exporte la représentation du modèle ODMG (cf. figure 4.5) ainsi que celle du schéma de la base courante. Le médiateur exporte la forêt qui décrit la structure de la vue intégrée. Elle est calculée par inférence de type à partir du programme YAT_L qui définit la vue (cf. section 5.2).

Accès aux opérations. Chaque source exporte l'ensemble des opérations qu'elle supporte dans le modèle du chapitre 6. Dans le cas d'un adaptateur, il s'agit des capacités sous-jacentes de la source, telles qu'elles ont pu être décrites dans le modèle opérationnel. Dans le cas d'un médiateur, il s'agit des opérations YAT complètes, auxquelles s'ajoutent les fonctions externes (que l'utilisateur peut définir comme il l'entend) et les opérations exportées à travers la vue. Notons que cette description inclut la liste des points d'entrée possibles dans les données (p.ex. noms des racines de persistance d'une base O_2 ou encore pages principales d'un site sur la Toile). Bien entendu, chaque composant se doit d'implanter les opérations qu'il exporte.

Accès aux données. Enfin, il est possible d'interroger les données, soit en soumettant une opération valide, soit en demandant l'évaluation d'un identifiant connu. Cette seconde possibilité s'apparente à de la navigation et peut se faire à partir des racines de persistance disponibles ou du résultat d'une opération précédente. Tout identifiant exporté par un composant YAT doit pouvoir être évalué par ce même composant. Il s'agit là d'une contrainte raisonnable dans la mesure où elle assure la cohérence de la vue et où elle est naturellement vérifiée par les sources existantes.

La YAT-API propose donc un ensemble de primitives simples. Celles-ci offrent néanmoins l'essentiel des fonctionalités que l'on trouve habituellement sur les systèmes de gestion de bases de données (exception faite des opérations de mises à jour). Elle sont donc suffisantes dans la pratique. Les informations structurelles exportées permettent au développeur une compréhension rapide des informations contenues dans les sources. Cela lui permet d'écrire plus facilement l'intégration qu'il souhaite. Enfin, la description des capacités opérationnelles permet de tirer parti au mieux des différents moyens d'interrogation.

7.2.3 Adaptateurs d'entrée

Le rôle des adaptateurs d'entrée est de donner la vue d'une source dans le modèle pivot. Plus un adaptateur est générique et plus il offre un bon rapport entre le temps de développement qu'il nécessite et le nombre d'applications pour lesquelles il est utile.

La figure 7.1 décrit en détails l'architecture d'un adaptateur. Celui-ci est composé de trois modules principaux, conçus pour rester le plus indépendant possible les uns des autres.

Module d'accès. Il résoud les problèmes d'interopérabilité et a la charge de faire correspondre les primitives de la YAT-API avec celles disponibles au niveau de la source. Il doit également implanter l'interface de communication comprise par la source. Par exemple, le module d'accès doit agir comme un client ODBC dans le cas d'une source relationnelle ou comme un client http dans le cas d'un site sur la Toile.

Module de conversion. Il sert à convertir les données de la source vers le modèle pivot. Dans le cas où la source utilise un format textuel, le module de conversion se compose principalement d'un parseur pour ce format. Le reste de la conversion s'effectue la plupart du temps trivialement, grâce à l'utilisation du modèle semistructuré.

Module de requête. Il doit être capable de transformer une requête, exprimée dans le modèle opérationnel YAT, en une expression du langage d'interrogation de la source. Écrire un tel module est parfois un travail délicat. L'utilisation judicieuse de la généricité permet heureusement de le réaliser une fois pour toutes.

Le reste de l'architecture des adaptateurs se compose d'une partie commune (absente de la figure) qui contient les modules ci-dessous.

Module de correspondance. Il sert à maintenir la relation qui existe entre les identifiants exportés vers le système YAT et les entités effectivement disponibles dans la source. Il

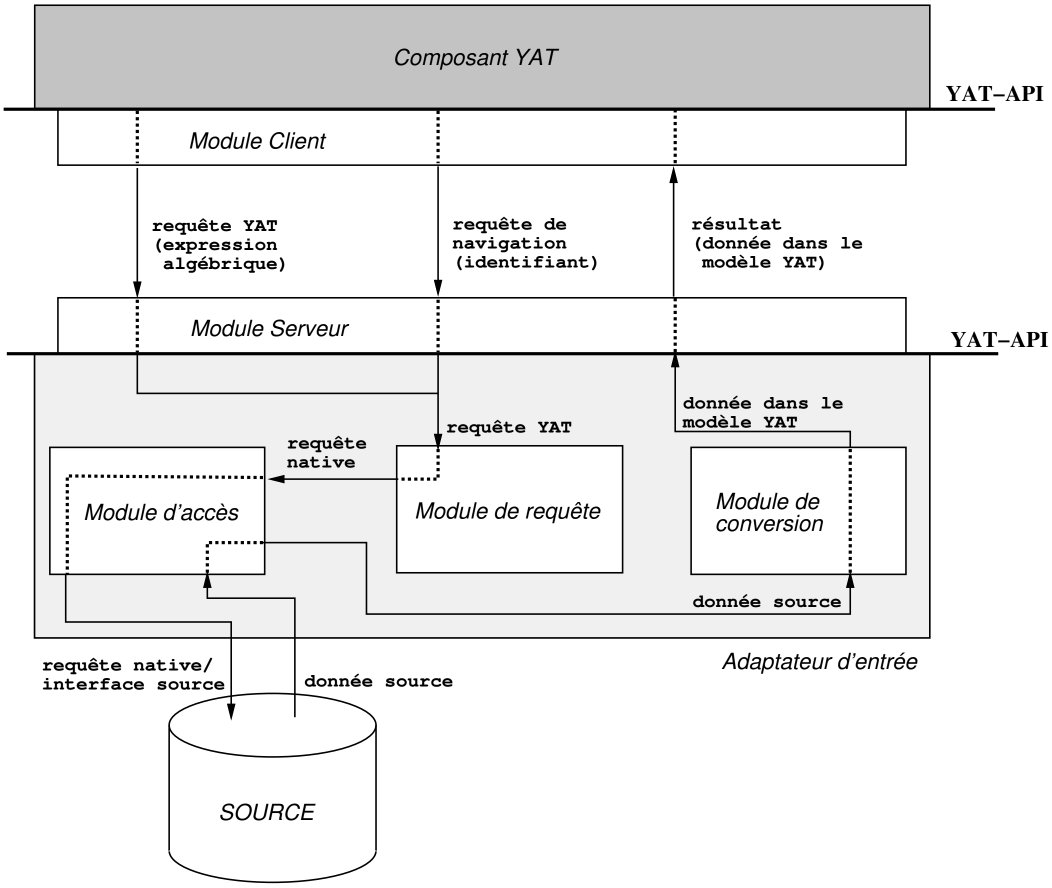


FIG. 7.1: *Architecture d'un adaptateur d'entrée*

implante donc une gestion d'identifiants externes. Remarquons que ce module n'est pas indispensable lorsque les identificateurs de la source sont externalisables (p.ex. URLs des pages d'un site de la Toile). Dans certains cas, il peut également être remplacé par un service similaire, lorsque celui-ci existe au niveau de la source (c'est par exemple le cas pour les dernières versions du SGBD O_2). Cette seconde option peut avoir un impact positif sur les performances, au prix d'un travail de développement supplémentaire.

Module principal. Un dernier module fait le lien entre l'ensemble des autres modules composant l'adaptateur. Il gère la communication entre ces derniers et implante également l'interface standard (YAT-API).

La figure 7.1 montre comment ces composants interagissent au cours de l'évaluation. En premier lieu, une requête est émise par un composant connecté à l'adaptateur. Celle-ci est exprimée en termes du modèle opérationnel. Elle est transmise au module de requête, qui la convertit en une requête compréhensible par la source, récupèrant au passage les identifiants dont il a besoin grâce à la table de correspondance. La source retourne alors un résultat, transmis à son tour au module de conversion, qui transforme les données dans le modèle pivot. Il doit prendre soin d'utiliser à nouveau la table de correspondance lors de la génération du résultat. Ce dernier est alors retourné au composant ayant réclamé l'information.

En soit, le développement de ces modules est rarement long, mais leur variété et leur nombre en fait une activité fastidieuse. Il est donc d'autant plus important de ne pas refaire plusieurs fois le même travail. C'est pourquoi l'ensemble des modules déjà implantés est organisé sous la forme d'une boite à outils. Ils peuvent rentrer dans la composition de nouveaux adaptateurs, ce qui assure leur réutilisation de manière optimale. Par exemple, un adaptateur pour un site de la Toile peut être développé en implantant un nouveau module de requêtes, mais en réutilisant le module de conversion HTML et le module d'accès pour les protocoles http/CGI [Gun96], déjà existants. Un adaptateur pour une base de données à objets différente d'O_2 peut être écrit en développant un nouveau module d'accès, mais en réutilisant le module de requête OQL et le module de conversion pour les objets ODMG.

État d'avancement. Un certain nombre d'adaptateurs d'entrée ont été réalisés (sans les modules de requêtes). Il s'agit d'un travail rarement difficile mais parfois long. Nous avons donc cherché à réutiliser au maximum le code existant. Notons que cette réutilisation a également été facilité par l'architecture de YAT. À l'heure actuelle, seuls les modules suivants sont disponibles.

- *Modules d'accès.*

  O_2. Le système de gestion de bases de données O_2 est fondé sur le modèle ODMG. Le module d'accès, baptisé `o2access`, offre un ensemble de services pour accéder à une base O_2 à travers le langage Caml. Il est possible de se connecter à une base, d'interroger le schéma (classes, types, racines de persistance) et les données (par navigation ou en transmettant une requête OQL).

  L'implantation de ce module est réalisée partiellement en Caml. Une passerelle en C est utilisée pour accéder à O_2 par son interface de programmation O_2API. Le choix de cette interface, plutôt que de la connection C++ ou Java, permet d'obtenir un module d'accès plus efficace et plus portable. Il a ainsi été testé avec succès sur les versions 4.5, 4.6 (sous SunOS) et 5.0 (sous Solaris) d'O_2. Notons que ce module est largement indépendant du système YAT et peut être utilisé pour d'autres applications écrites en Caml.

  http. Ce module d'accès permet de récupérer des données à travers le protocole http. Il est conçu au-dessus du client http fournit avec V6 [LR96], et développé par François ROUAIX.

- *Modules de conversion.*

  ODMG. Le module ODMG permet de convertir les données contenues dans une base de données à objets dans le modèle YAT. En conjonction avec le module d'accès O_2, ce module constitue l'adaptateur O_2.

  HTML. Le module de conversion HTML est bâti autour du parseur HTML écrit par François ROUAIX pour le brouteur MMM [Rou96].

  XML. Le module de conversion XML est bâti sur Tony [3], le parseur XML écrit par Christian LINDIG.

  BIBTEX. Ce module est quant à lui bâti au dessus du parseur pour le format BIBTEX développé par Jean-Christophe FILLIÂTRE [4].

3. `http://www.cs.tu-bs.de/softech/people/lindig/software/tony.html`
4. `http://www.lri.fr/~filliatr/bibtex2html/index.en.html`

hand. Grâce à ce module, il est possible d'accéder aux données stockées dans une base `hand` (p.ex. celle qui contient les informations sur les personnels de l'INRIA).

Object Container. Il s'agit d'un module développé dans le cadre du projet OPAL. Les « Object Containers » permettent le transport des données utilisées par la plate-forme OPAL. Il s'agit d'une représentation interne utilisée pour des données obéïssant au modèle STEP/EXPRESS, donc proche du modèle objet. Ce module est partiellement écrit en C++ pour permettre l'accès aux librairies définissant les Object Containers.

7.2.4 Médiateurs

Le médiateur est le composant situé au cœur du système. Une lourde tâche lui incombe : celle de réaliser l'intégration structurelle à partir des informations fournies par les sources. Il doit répondre aux requêtes formulées par les applications dans un temps raisonnable et s'avère souvent être le goulot d'étranglement de tout le système. Par la même occasion, il offre de nombreuses opportunités en matière d'amélioration des performances. C'est donc tout naturellement que nous nous sommes intéressés, dans le chapitre précédent, aux techniques d'optimisation susceptibles d'y être implantées.

Le médiateur est organisé autour du tandem constitué par un module d'évaluation et un module d'optimisation. La figure 7.2 présente plus en détails l'architecture interne d'un médiateur YAT.

Module d'optimisation. Ce module implante les techniques d'optimisation mises au point dans le chapitre 6. Étant donnée une requête, il exploite l'ensemble des connaissances dont il dispose pour trouver une bonne stratégie d'évaluation. La spécification de l'intégration en YAT_L (conservée comme un ensemble d'expressions algébriques) sert, entre autres, à déterminer quelles sont les sources à accéder pour répondre à la requête. Les informations sur les capacités respectives des sources permettent de répartir précisément l'évaluation entre les différents composants. Le résultat de l'optimisation est un plan d'exécution distribué correspondant à une évaluation efficace.

Module d'évaluation. Le rôle du module d'évaluation est d'appliquer le plan d'exécution. Cela consiste à transmettre les sous-requêtes spécifiques à chacune des sources impliquées, à récupérer les résultats partiels, puis à achever l'évaluation localement avant de retourner le résultat. Notons que la dernière étape peut nécessiter l'application des fonctions de Skolem, maintenues localement par le médiateur. Il est important de remarquer que le module d'évaluation a l'opportunité d'exploiter le parallelisme des différentes sources. On peut imaginer qu'il envoie simultanément les requêtes aux sources. Dans ce cas, il doit également assurer la synchronisation entre les différents processus.

Quelques modules du médiateur ne sont pas visibles sur la figure 7.2. Le **module de correspondance** s'occupe de l'évaluation des fonctions de Skolem tandis que le **module principal** encapsule le tout et implante la YAT-API. Le **module YAT_L** est utilisé lors de l'initialisation du médiateur pour générer la représentation algébrique définissant la vue à partir du programme d'intégration. Ce module contient un parseur pour YAT_L et un convertisseur algébrique « naïf ».

Pour finir, expliquons comment le médiateur traite les requêtes qui lui sont envoyées. Le processus d'évaluation est illustré sur la figure 7.2. La requête est tout d'abord transmise au

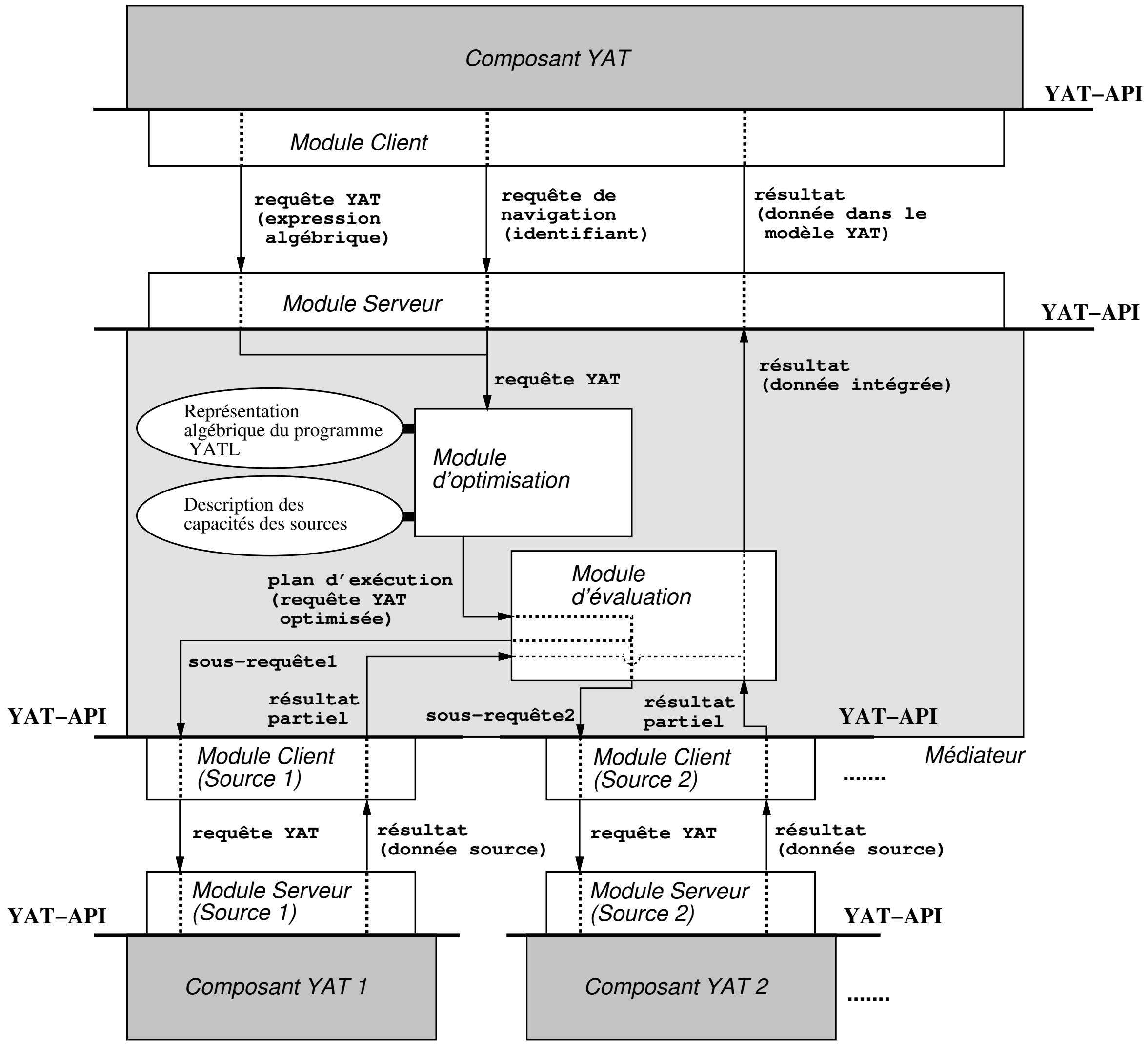


FIG. 7.2: *Architecture d'un médiateur et fonctionnement pendant l'évaluation*

module d'optimisation. Elle est alors appariée avec les expressions algébriques qui définissent la vue, fournissant un premier plan d'exécution évaluable sur les sources. C'est alors que l'optimisation prend place, suivant les trois phases que nous avons décrites dans la section 6.4. Il en résulte un plan d'exécution optimisé qui indique notamment quelles sous-requêtes doivent être évaluées par les sources. Ce plan d'exécution est envoyé au module d'évaluation qui, comme nous l'avons déjà vu, transmet les sous-requêtes aux sources, récupère les résultats partiels et achève l'évaluation.

État d'avancement. Le médiateur YAT contient pour le moment le module YAT_L ainsi que le module d'évaluation. Il propose donc toutes les fonctionnalités nécessaires à l'écriture de programmes d'intégration. Il est implanté comme un interprète pour le langage, ce qui permet de ne jamais avoir à recompiler le médiateur. La syntaxe de YAT_L comprise par le parseur est celle données dans l'annexe A.

L'optimiseur n'étant pas encore implanté, il peut être utilisé soit quand les données d'entrées tiennent en mémoire, soit dans un contexte où l'utilisateur peut facilement déterminer les requêtes à envoyer aux sources. Sur des sources plus volumineuses, il est indispensable de terminer l'implantation de l'optimiseur.

7.2.5 Composants applicatifs

Pour achever la présentation de l'architecture, nous décrivons sommairement la partie applicative du système. Bien entendu, il n'est pas question ici de répondre à l'ensemble des besoins rencontrés par les applications. Cependant, le système YAT propose en standard quelques outils couramment utilisés en pratique. Parmi ceux-ci, on trouve : (i) des interfaces pour accéder de façon générique aux composants YAT et (ii) des adaptateurs de sortie permettant d'exporter les données vers un format cible.

Interfaces génériques. Elles sont fournies par le système et donnent à l'utilisateur la possibilité d'interagir avec une source YAT quelconque (*i.e.* un composant qui implante la YAT-API). Une fois l'interface connectée, il est possible d'envoyer une requête, de naviguer à travers les données, etc. Comme on peut le voir sur la figure 7.3, l'implantation de ces composants est très simple.

- L'interface de navigation (à droite de la figure) se compose d'un module unique. Ce module est seulement capable de demander l'évaluation d'un identifiant et d'afficher le motif correspondant. En itérant ce processus à partir des points d'entrées fournis par la source, cette interface donne la possibilité de naviguer dans le graphe des données. De fait, elle permet de naviguer dans n'importe quelle forêt YAT et peut donc également être utilisée pour accéder aux informations sur la structure.

- L'interface opérationnelle (qui n'est pas décrite sur la figure) permet de prendre connaissance des opérations disponibles au niveau de la source ainsi que de rajouter de nouvelles équivalences (cf. section 6.3.2).

- L'interface de requête (à gauche de la figure) donne la possibilité de soumettre des requêtes formulées en YAT_L. Elle est constituée d'une interface d'affichage simple et du module YAT_L (cf. section précédente), qui parse la requête et la transcrit dans l'algèbre.

Remarquons que les requêtes sont communiquées directement sous la forme d'expressions algébriques entre les différents composants du système. Un tel choix permet de rester le plus indépendant possible du langage d'interrogation. Il est ainsi très simple de proposer un autre langage à partir du moment où il est capturé par l'algèbre (p.ex. Lorel sans expressions de chemin généralisées). De plus, cela facilite la définition de la YAT-API, comme une interface homogène entre adaptateurs et médiateurs.

Remarquons également que l'existence de cette API commune rend ici de nombreux services. Par exemple, l'interface de navigation peut être employée indépendamment sur les adaptateurs (pour prendre connaissance des structures lors de la phase d'intégration, pour naviguer à la main dans les données) ou sur les médiateurs (pour prendre connaissance des structures lors du développement d'une application, pour naviguer dans la vue intégrée). L'interface opérationnelle est identique, qu'il s'agisse des adaptateurs ou des médiateurs.

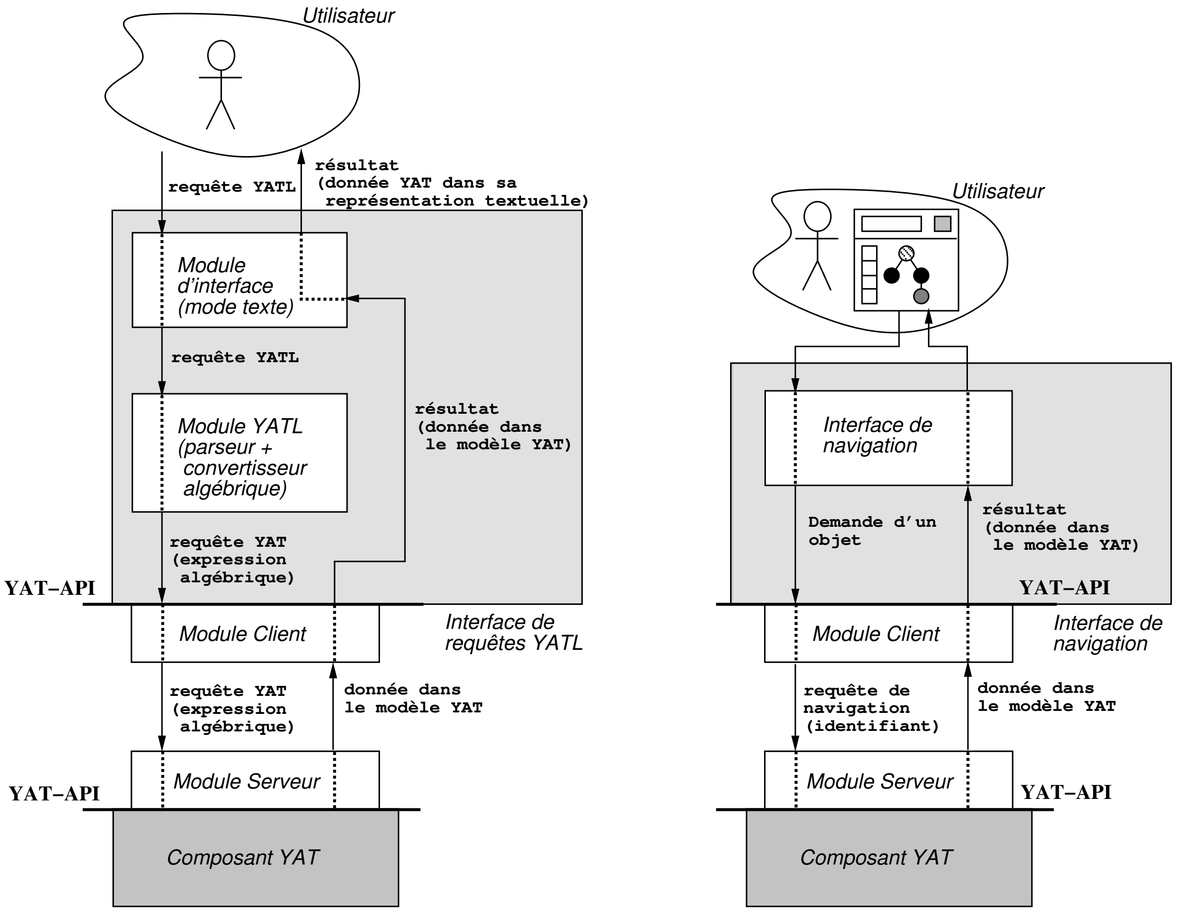


FIG. 7.3: *Interfaces pour l'interrogation et la navigation*

Adaptateurs de sortie. Les adaptateurs de sortie servent à exporter la vue intégrée vers un format et un système cibles. Ils permettent ainsi à l'application d'accéder aux données dans la représentation qui lui convient. Il s'agit là d'une fonction importante, souvent négligée dans les autres systèmes d'intégration. Les services offerts par ces composants peuvent être très variables : simple convertisseur (p.ex. générant des fichiers BIBTEX), implantant un protocole d'accès particulier (p.ex. CGI pour un adaptateur de sortie qui génère des pages HTML) ou encore permettant de peupler une base de données.

Les adaptateurs de sortie jouent un rôle symétrique à celui des adaptateurs d'entrée. Ils se fondent notamment sur la même représentation des données, ce qui signifie que seuls les motifs ayant la structre adéquate pourront être exportés (p.ex. motifs ODMG vers une base O_2). Pour cette raison, les mécanismes de vérification de type sont essentiels dans ce contexte, car ils assurent que les données intégrées auront bien la structure requise.

À titre d'exemple, la figure 7.4 donne l'architecture d'un adaptateur de sortie relationnel, tel qu'il pourrait être implanté. Il se compose de modules remplissant des fonctions similaires (à l'envers !) à ceux d'un adaptateur d'entrée. On retrouve donc :

– un **module d'accès**, qui implante cette fois-ci le côté serveur du protocole ODBC ;

– un **module de requête**, qui parse les requêtes SQL qui lui sont envoyées pour les

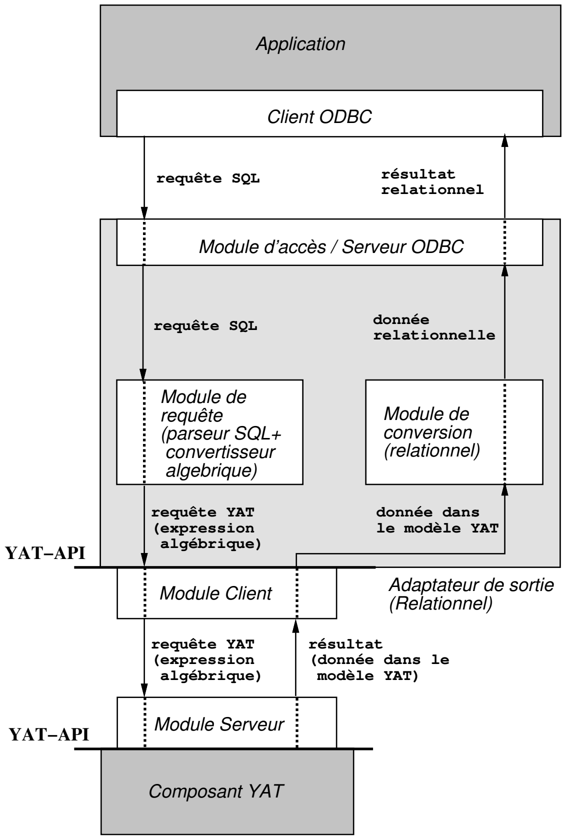


FIG. 7.4: *Un adaptateur de sortie relationnel*

transcrire dans l'algèbre YAT ;

– un **module de conversion**, qui transforme le résultat d'une requête du modèle YAT vers le modèle relationnel.

Notons que le module de requête doit tenir compte des annotations additionnelles, ajoutées lors la conversion du relationnel en YAT. En effet, ces annotations doivent impérativement figurer dans les filtres utilisés comme paramètres dans les opérations *Bind* et *Tree*. Il est donc important de rester attentif à la cohérence entre le module de conversion et le module de requête. Notons également que la conversion vers le relationnel ne peut avoir lieu que si les données retournées par le composant YAT correspondent bel et bien à une structure relationnelle. Cela démontre, une fois de plus, l'utilité des vérifications de type. Enfin, il faut également que les informations de type fournies par le composant YAT sous-jacent suffisent au module de conversion pour obtenir le schéma relationnel correspondant (qui peut être interrogé par l'application *via* l'interface ODBC).

État d'avancement. Plusiers modules rentrant dans la composition d'adaptateurs de sortie ont été développés.

– *Modules d'accès.*

CGI. Ce module permet d'interroger un composant YAT qui retourne des pages HTML.

Il prend en entrée l'identifiant d'un motif et retourne la page correpondant à ce motif.

Système de fichier Ce module permet de matérialiser des données dans des fichiers Unix. Une particularité est qu'il peut appeler indistinctement tous les modules de conversion qui génèrent du texte afin d'obtenir le contenu d'un fichier.

– *Modules de conversion.*

HTML. Un module pour HTML permet de générer des pages à partir de motifs ayant une structure appropriée. Ce module implante la version 3.2 de la norme HTML [Rag97].

Gnuplot. Un module permet également de générer des fichiers compréhensibles par l'outil de création de graphiques, gnuplot.

7.3 Utilisation du système

7.3.1 Mode d'emploi

À présent, expliquons de quelle façon le système YAT est utilisé lors de la phase d'intégration. Les étapes suivantes ne sont certainement pas obligatoires, mais elles correspondent à une utilisation « normale » du système.

étape n° 1. *Phase de développement.*

Une fois que les différentes sources participant à l'intégration sont déterminées, il se peut que certains adaptateurs ne soient pas fournis par le système. Il faut donc tout d'abord développer les modules manquants.

étape n° 2. *Installation des composants.*

Il faut ensuite connecter les adaptateurs correspondants à chaque source, puis faire de même avec le médiateur. Durant cette étape, l'utilisateur doit fournir les paramètres nécessaires à l'initialisation. Un composant applicatif final (adaptateur de sortie) peut également être installé.

étape n° 3. *Spécification de l'intégration.*

Il faut écrire le programme d'intégration en YAT$_L$. L'utilisateur peut exploiter les informations structurelles fournies par les adaptateurs et employer les mécanismes de contrôle de type pour vérifier ses programmes. Il est également possible de réutiliser un programme déjà écrit (souvent générique) ou de générer automatiquement un programme par instanciation (en usant des mécanismes décrits dans la section 5.3).

étape n° 4. *Personnalisation des capacités des sources.*

Enfin, il est possible de procéder à des ajustements sur les opérations disponibles. On peut par exemple, exporter une opération des sources vers la vue (voir section 6.1). On peut également rajouter une équivalence algébrique pour rendre certaines optimisations possibles (cf. section 6.3.2).

L'étape de développement est bien entendu la plus coûteuse. Dans le meilleur des cas, tous les composants logiciels nécessaires (adaptateurs d'entrée et de sortie) sont déjà présents. L'intégration se réduit alors à sa plus simple expression, consistant uniquement à écrire le programme YAT$_L$ souhaité. Grâce à l'utilisation du modèle semistructuré, nous avons vu qu'il était possible de réaliser des adaptateurs génériques. Dans le cas où il est nécessaire d'écrire de nouveaux modules, ceux-ci viennent enrichir le système. Ainsi, le nombre de cas impliquant une phase de développement diminue à chaque nouveau module produit. On voit clairement comment cela permet de se rapprocher toujours plus du cas de figure idéal, dans lequelle l'étape n^o 1 a complètement disparu.

7.3.2 Quelques applications

Pour conclure, nous évoquons quelques-unes de applications ayant été développées avec YAT. Pour chacune d'entre elles, il a souvent été nécessaire d'implanter un ou plusieurs modules pour les adaptateurs. C'est pourquoi le YAT supporte désormais un nombre raisonnable de formats et de systèmes sources.

Une application de conversion : de O_2 vers gnuplot. L'objectif de cette application était de pouvoir accéder à des information statistiques (concernant le résultat de batteries de tests pour OQL) afin de générer des courbes de comparaisons. En effet, la quantité importante des données rend l'interprétation difficile sans l'aide d'outils graphiques. Cette application a été réalisée en collaboration avec Fanny WATTEZ.

Il s'agit là d'une simple application de conversion, impliquant les composants suivants :

- le médiateur YAT;
- l'adaptateur d'entrée pour O_2, qui permet d'envoyer des requêtes OQL ;
- l'adaptateur de sortie pour le système de fichier, qui utilise également le module de conversion pour gnuplot.

Un module principal très simple permet de faire le lien entre ces composants, qui sont linkés et n'utilisent donc pas l'architecture client-serveur de YAT. Ce module principal sert à parser la ligne de commande, envoie la requête OQL nécessaire à O_2, puis lance le processus de conversion. Le programme YAT$_L$ utilisé est très simple. Il est capable de filtrer le résultat de la requête OQL puis de générer le motif gnuplot approprié. Tout le reste est pris en charge par le système. À titre indicatif, la taille de ce programme principal est de 150 lignes de code. La réalisation de l'application (y compris l'adaptateur gnuplot et l'adaptateur pour le système de fichiers) a pris moins d'une journée.

Exploiter la généricité : une implantation alternative d'O_2Web. Cette deuxième application démontre la capacité du système YAT à effectuer des opérations à différents degrés de généricité. Nous avons réimplanté O_2Web avec YAT.

Il s'agit d'une application de conversion avec un programme générique, impliquant les composants suivants :

- le médiateur YAT;
- l'adaptateur d'entrée pour O_2 ;

– l'adaptateur de sortie HTML/CGI.

Cette application démontre également l'intérêt d'utiliser YAT pour personnaliser une telle application. Par exemple, il est très simple de rajouter une règle spécifique au programme d'intégration, afin d'obtenir une transformation appropriée au contenu de la base. Par défaut, le programme générique employé fonctionne sur n'importe qu'elle base O_2. Cette application peut être consultée sur la Toile à l'addresse suivante : `http://www-rocq.inria.fr/~simeon/YAT/o2web.html`

Création d'un site sur la Toile à partir de sources hétérogènes Pour finir, YAT a été utilisé pour la mise en place d'une partie du serveur de l'équipe Verso [SC98]. L'objectif de cette application était de répondre aux besoins classiques lors de la réalisation de site sur la Toile, c'est à dire de pouvoir accéder à l'ensemble des informations dans une ou plusieurs sources, d'éviter la duplication d'information (pour faciliter la mise à jour), de proposer la même information sous plusieurs formes et de pouvoir y accéder par plusieurs moyens (listes complète, accès par mot clé, etc).

Nous avons stocké les informations sur les chercheurs de l'équipe dans une bases de données O_2 et les informations sur les publications dans un document XML. Le document est indexé avec Wais, ce qui permet d'interroger par mots-clés.

Il s'agit donc d'une application d'intégration réalisée avec les composants suivants :

– le médiateur YAT;

– l'adaptateur d'entrée pour O_2 ;

– l'adaptateur d'entrée pour XML ;

– l'adaptateur de sortie HTML/CGI.

Le site HTML est construit à partir des informations de nos deux sources en utilisant YAT$_L$. Les pages sont matérialisées à la demande, c'est-à-dire que les pages sont générées à chaque requête de l'utilisateur. Le problème principal est donc de ne pas récupérer l'ensemble des informations des sources à chaque fois. Le programme principal a donc la charge d'envoyer la requête qui correspond aux informations utiles pour une page donnée. Cela reste relativement simple, dans la mesure où ces requêtes sont : soit récupérer toute les publications (pour générer une liste intégrale), soit accéder par mot clé (en utilisant Wais).

L'utilisation d'un langage déclaratif tel YAT$_L$ permet de multiplier rapidement le nombre de pages générées. Les primitives de regroupement et d'ordonnancement sont également très utiles dans le contexte de HTML. Le site propose ainsi des listes triées par numéro de publication, regroupées par nom d'auteur, par années, etc. Ajoutons que l'implantation générique du médiateur (plutôt qu'une approche par compilation ou génération de code) permet de rajouter très facilement des règles d'intégration sans toucher à l'installation logicielle.

Cette application peut être consultée sur la Toile à l'addresse suivante : `http://www-rocq.inria.fr/~simeon/YAT/publis.html`

Chapitre 8

Perspectives

Insufficient facts always invite danger.
– Spock
(in *Space Seed*, stardate 3141.9)

Dans cette thèse, nous avons développé un ensemble de techniques qui facilitent l'installation de composants logiciels effectuant des tâches d'intégration. Nous avons également décrit un système qui implante ces techniques et permet de réaliser des applications accédant aux données de sources hétérogènes et distribuées.

Dans [ACM97], les auteurs plaidaient en faveur de l'utilisation d'un modèle de données semistructuré et d'un langage d'intégration déclaratif comme fondations pour la manipulation de données hétérogènes. Dans un premier temps, nous avons étendu ce travail dans trois directions.

- Nous avons élaboré un système de types pour le modèle semistructuré. Grâce au mécanisme d'instanciation, celui-ci permet la description rigoureuse des structures de données de sources hétérogènes, tout en conservant un maximum de flexibilité.

- Nous avons adapté le langage d'intégration proposé dans [ACM97] afin de répondre aux exigences pratiques des applications et simplifier la spécification des programmes. Ce langage, baptisé YAT$_L$, fournit des primitives de haut niveau pour la création d'objets et le traitement des collections. De plus, il offre des possibilités de vérification, fondées sur le typage et des mécanismes permettant d'automatiser partiellement l'écriture de nouveau programmes.

- Nous avons développé un système d'intégration qui utilise ces techniques. Un prototype a été implanté dans le cadre du projet européen OPAL. Il a démontré son aptitude à servir de support à la réalisation rapide d'applications impliquant l'accès à des sources hétérogènes.

La représentation des informations de structure dans un modèle commun ainsi que l'utilisation d'un langage d'intégration déclaratif, ouvrent la voie à l'optimisation. Dans un deuxième temps, nous avons mis au point des techniques pour l'évaluation efficace de requêtes posées par les applications.

- Nous avons introduit un modèle opérationnel et une algèbre qui servent de support à l'évaluation et à l'optimisation des requêtes portant sur la vue intégrée. Les réécritures définies sur l'algèbre permettent de retrouver les techniques d'optimisation des systèmes structurés et d'en concevoir de nouvelles, adaptées au contexte semistructuré.

– Les propriétés de généricité du typage sont de nouveau utilisées pour obtenir une description précise des capacités opérationnelles des sources. Cela permet de retrouver également les techniques d'optimisation distribuées. On peut alors déterminer une politique d'évaluation plus performante, répartie entre les composants du système d'intégration et les sources.

La principale originalité de l'approche réside dans l'utilisation d'un modèle de données possédant des propriétés de généricité. Cette caractéristique est exploitée à tous les niveaux du système d'intégration. Il en résulte une grande flexibilité, qui permet de s'adapter précisément aux besoins des applications, à l'hétérogénéité des sources et aux modifications de l'environnement d'intégration (ajout d'une source, évolution de schéma, etc). Grâce à l'ensemble de ces techniques, il est enfin possible de (ré)concilier les fonctionnalités, jusqu'à présent antagonistes, des systèmes d'intégration existants. Le système YAT peut procéder à une intégration dans un temps record, tout en conservant de bonnes performances lors de la manipulation de grandes quantités d'informations. Nous démontrons ainsi la faisabilité d'un système alliant la simplicité de l'approche semistructurée à l'efficacité de l'approche structurée.

Nous ne prétendons pas pour autant fournir une solution complète et définitive à l'intégration : de nombreuses questions restent encore sans réponse. Nous croyons cependant que l'une des contributions de ce travail est d'avoir distingué les différents problèmes mis en jeu. En séparant clairement le traitement de ces problèmes, nous espérons que l'approche proposée pourra aisément servir de point de départ à de nouveaux travaux. Dans cette optique, nous avons sélectionné trois pistes qui nous semblent prometteuses : l'amélioration des techniques d'optimisation, l'étude poussée des problèmes sémantiques et un investissement dans l'implantation d'un système plus robuste.

Optimisation

Les techniques d'optimisation proposées dans cette thèse ne représentent à notre avis qu'un bon point de départ, certes mis en valeur par l'absence d'optimisation patente dans les systèmes semistructurés. Parmi toutes les opportunités d'optimisation qui restent à explorer, en voici quelques-unes qui ont attiré notre attention.

– L'élaboration d'un modèle de coût devrait permettre d'exploiter plus avant les techniques présentées ici. Intégrer les coûts des opérateurs supportés par les sources dans la stratégie de recherche peut servir à répartir l'évaluation de façon optimale entre les sources (à la manière de DISCO [NGT98]).

– Les expressions de chemin généralisées représentent un paradigme d'interrogation fort intéressant du point de vue de l'utilisateur. Elles fournissent un moyen élégant et déclaratif de mélanger interrogation structurée et non structurée. Du point de vue du langage, il serait simple de rajouter de telles expressions à YAT_L. Malheureusement, il est très facile d'écrire une requête impliquant un parcours complet de la base de données, voire qui la retourne en résultat ! L'évaluation des requêtes dans un tel contexte est un véritable défi. Nous pensons néanmoins que les capacités de typage de YAT et l'utilisation d'index plein-texte devraient lui permettre de tirer son épingle du jeu. Certains travaux récents sur le sujet sont encourageants (voir [Chr96, CCM96, CCMS98, FPS97, BM99]).

Sémantique

Nous n'avons fait que survoler les problèmes liés à la sémantique des données. Plusieurs systèmes offrent des fonctionnalités plus avancées en la matière, comme Information Manifold [LRO96] qui utilise des logiques descriptives [Bor95] ou dans une moindre mesure, Garlic [CHS+95] qui utilise un modèle orienté-objet. Ces systèmes autorisent des descriptions sémantiques plus riches que YAT. Incorporer davantage de fonctionnalités sémantiques dans un système semistructuré reste un sujet d'étude. Pour donner une idée de l'ampleur du problème, voici en vrac quelques questions pour lesquelles nous serions heureux d'avoir des réponses...

- Comment intégrer les autres formes d'information sémantique (méta-données, ontologies, thésauri, etc) offertes par chaque source?
- Comment ces informations interagissent-elles avec la structure des données?
- Comment tirer parti d'une connaissance sur le contenu respectif des sources pour offrir à l'utilisateur la meilleure qualité d'information possible?
- Les informations sémantiques supplémentaires ouvrent-elle de nouvelles opportunités d'optimisation?
- Dans YAT, l'utilisateur se sert de la connaissance sémantique qu'il possède sur les sources lors de l'écriture des programmes d'intégration. Ce travail peut-il être partiellement automatisé lorsque certaines de ces informations sémantiques sont connues par le système?

Prototype

Un travail d'implantation important reste à fournir sur le système YAT. Il offre des fonctionnalités encore réduites comparé à ses concurrents directs, tels TSIMMIS [PGW95, PAG96], DISCO [TRV96], IRO-DB [GGF+95] ou encore Garlic [CHS+95]. Dans tous les cas, la réalisation de l'optimiseur fondé sur les techniques décrites dans le chapitre 6 constitue une première étape indispensable. Les développements ultérieurs pourront ensuite être poursuivis dans plusieurs directions.

- La mise au point d'un module de stockage est à l'étude. Un tel composant rendrait possible l'utilisation du système dans un contexte matérialisé ou partiellement matérialisé. De plus, il permettrait l'étude des problèmes spécifiques à ce type de configuration, comme ceux relatifs à l'optimisation du chargement [Ame99] ou à la propagation des mises à jours [CH95, HZ96, ABS97]. L'utilisation des fonctions de Skolem dans la définition de l'intégration devrait alors se révéler d'une aide appréciable.
- Du fait de leur généricité, le développement de nouveaux adaptateurs ou de composants applicatifs (adaptateurs de sortie vers différents formats, interfaces pour les langages de requêtes utilisateur) enrichissent le système en permanence. Seuls quelques sources ou

formats sont supportés à l'heure actuelle : le développement de nouveaux composants pourra être entrepris.

– Enfin, une distribution publique du système YAT est envisagée dans un proche avenir. Dans ce but, il serait souhaitable d'améliorer les interfaces avec le système et de fournir une documentation plus complète.

Bibliographie

[AB91] Abiteboul (S.) et Bonner (A. J.). – Objects and Views. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 238–247. – Denver, Colorado, mai 1991.

[AB95] Abiteboul (S.) et Beeri (C.). – The Power of Languages for the Manipulation of Complex Values. *The VLDB Journal*, vol. 4, n° 4, octobre 1995, pp. 727–794.

[ABCD+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.

[Abi97] Abiteboul (S.). – Querying Semi-Structured Data. *In : Proceedings of International Conference on Database Theory (ICDT)*, pp. 1–18. – Delphi, Greece, janvier 1997.

[ABS97] Amer-Yahia (S.), Brèche (P.) et Souza dos Santos (C.). – Object views and updates. *Ingénierie des Systèmes d'Information*, vol. 5, n° 1, avril 1997, pp. 63–89.

[ACC+97] Abiteboul (S.), Cluet (S.), Christophides (V.), Milo (T.), Moerkotte (G.) et Siméon (J.). – Querying Documents in Object Databases. *International Journal on Digital Libraries*, vol. 1, n° 1, avril 1997, pp. 5–19.

[ACD98] Amer-Yahia (S.), Cluet (S.) et Delobel (C.). – Bulk Loading Techniques for Object Databases and an Application to Relational Data. *In : Proceedings of International Conference on Very Large Databases (VLDB).* – New York, août 1998.

[ACHK93] Arens (Y.), Chee (C. Y.), Hsu (C.-N.) et Knoblock (C. A.). – Retrieving and integrating data from multiple information sources. *International Journal of Cooperative Information Systems*, vol. 2, n° 2, 1993, pp. 127–158.

[ACM93] Abiteboul (S.), Cluet (S.) et Milo (T.). – Querying and Updating the File. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 73–84. – Dublin, Ireland, août 1993.

[ACM95] Abiteboul (S.), Cluet (S.) et Milo (T.). – A Database Interface for File Updates. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 386–397. – San Jose, California, mai 1995.

[ACM97] Abiteboul (S.), Cluet (S.) et Milo (T.). – Correspondence and Translation for Heterogeneous Data. *In : Proceedings of International Conference on Database Theory (ICDT)*, pp. 351–363. – Delphi, Greece, janvier 1997.

[ACM98] Abiteboul (S.), Cluet (S.) et Milo (T.). – A Logical View of Structured Files. *The VLDB Journal*, vol. 7, n° 2, mai 1998, pp. 96–114.

[ADD+91] Ahmed (R.), De Smedt (P.), Du (W.), Kent (W.), Ketabchi (M. A.), Litwin (W.), Rafii (A.) et Shan (M.-C.). – The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, vol. 24, n° 12, décembre 1991, pp. 12–18.

[AG87] Abiteboul (S.) et Grumbach (S.). – Databases and structured objects. *Technology and Science of Informatics*, vol. 6, n° 8, 1987, pp. 577–595.

[AGVW89] Abiteboul (S.), Grumbach (S.), Voisard (A.) et Waller (E.). – An extensible rule-based language with complex objects and data-functions. *In : Proceedings of International Workshop on Database Programming Languages*, pp. 298–314. – Gleneden Beach, Oregon, juin 1989.

[AH88] Abiteboul (S.) et Hull (R.). – Restructuring hierarchical database objects. *Theoretical Computer Science*, vol. 62, 1988, pp. 3–38.

[AHK95] Arens (Y.), Hull (R.) et King (R.). – *Reference Architecture for the Intelligent Integration of Information.* – Rapport technique n° Version 2.0 (draft), Advanced Research Project Agency, août 1995.
`http://mole.dc.isx.com/I3/`.

[AHV95] Abiteboul (S.), Hull (R.) et Vianu (V.). – *Foundations of Databases.* – Addison-Wesley, 1995.

[AK89] Abiteboul (S.) et Kanellakis (P.). – Object identity as a query language primitive. *In : Proceedings of ACM SIGMOD Conference on Management of Data.* ACM, pp. 159–173. – Portland, Oregon, juin 1989.

[AM98] Arocena (G.) et Mendelzon (A.). – Weboql: Restructuring documents, databases, and webs. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE).* – Orlando, février 1998.

[Ama94] Amann (B.). – *Interrogation d'hypertextes.* – Thèse de doctorat, Conservatoire National des Arts et Métiers, février 1994.

[Ame97] Amer-Yahia (S.). – From relations to objects: The reloo prototype. *In : XIII congrès BDA, Journées O_2 Technology*, pp. 56–78. – Grenoble, France, septembre 1997.

[Ame99] Amer-Yahia (S.). – *Du Chargement en Masse dans une Base de Données (en Général) et de la Migration Relationnel-Objet (en particulier).* – Thèse de doctorat, Université Paris XI, janvier 1999.

[AMM97] Atzeni (P.), Mecca (G.) et Merialdo (P.). – To weave the web. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 206–215. – Athens, Greece, août 1997.

[AMM98] Atzeni (P.), Mecca (G.) et Merialdo (P.). – Design and maintenance of data-intensive web sites. *In : Proceedings of the International Conference on Extending Database Technology*, pp. 436–450. – Valencia, Spain, mars 1998.

[Amo94] Amor (R.). – *A Mapping Language for Views.* – Rapport technique, New Zealand, Computer Science Department, University of Auckland, 1994.

[AMR+98] Abiteboul (S.), McHugh (J.), Rys (M.), Vassalos (V.) et Wiener (J. L.). – Incremental maintenance for materialized views over semistructured data. *In : Proceedings of International Conference on Very Large Databases (VLDB).* – New-York, août 1998.

[ANS95] ANSI/NISO. – *Z39.50-1995 (Versions 2 and 3) Information Retrieval: Application Service Definition and Protocol Specification.* – 1995.

[AQM+97] Abiteboul (S.), Quass (D.), McHugh (J.), Widom (J.) et Wiener (J. L.). – The lorel query language for semistructured data. *International Journal on Digital Libraries*, vol. 1, n° 1, avril 1997, pp. 68–88.

[ASN94] *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.* – ITU-T Recommendation X.680, juillet 1994. ISO/IEC 8824-1:1994 (E).

[AV97] Abiteboul (S.) et Vianu (V.). – Regular path queries with constraints. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 122–133. – Tucson, Arizona, mai 1997.

[Bai95] Bailey (I.). – *EXPRESS-M Reference Manual.* – CIMIO Ltd, juillet 1995. ISO TC184/ SC4/WG5 N243.

[BBB+97] Bayardo (R.), Bohrer (W.), Brice (R.), Cichocki (A.), Fowler (G.), Helal (A.), Kashyap (V.), Ksiezyk (T.), Martin (G.), Nodine (M.), Rashid (M.), Rusinkiewicz (M.), Shea (R.), Unnikrishnan (C.), Unruh (A.) et Woelk (D.). – Infosleuth: Semantic Integration of Information in Open and Dynamic Environments. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 195–206. – Tucson, Arizona, mai 1997.

[BC81] Bernstein (P. A.) et Chiu (D. W.). – Using semi-joins to solve relational queries. *Journal of the ACM*, vol. 28, n° 1, janvier 1981, pp. 25–40.

[BC92] Belki (N. J.) et Croft (W. B.). – Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, vol. 35, n° 12, décembre 1992, pp. 29–38.

[BDFS97] Buneman (P.), Davidson (S. B.), Fernandez (M.) et Suciu (D.). – Adding Structure to Unstructured Data. *In : Proceedings of International Conference on Database Theory (ICDT).* – Delphi, Greece, janvier 1997.

[BDH+95] Buneman (P.), Davidson (S. B.), Hart (K.), Overton (C.) et Wong (L.). – A data transformation system for biological data sources. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 158–169. – Zurich, Switzerland, septembre 1995.

[BDHS96] Buneman (P.), Davidson (S. B.), Hillebrand (G.) et Suciu (D.). – A query language and optimization techniques for unstructured data. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 505–516. – Montreal, Canada, juin 1996.

[BDK92] Bancilhon (F.), Delobel (C.) et Kanellakis (P.). – *Building an Object-Oriented Database System: The Story of O_2.* – San Francisco, CA, Morgan Kaufmann, 1992.

[BDS95] Buneman (P.), Davidson (S. B.) et Suciu (D.). – Programming constructs for unstructured data. *In : Proceedings of International Workshop on Database Programming Languages.* – Gubbio, Italy, 1995.

[BEP+91] Blaha (M.), Eddy (F.), Premerlani (W.), Lorensen (W.) et Rumbaugh (J.). – *Object Oriented Modeling and Design.* – Prentice-Hall International Editions, 1991.

[Ber92] Bertino (E.). – A view mechanism for object-oriented databases. *In : Proceedings of the International Conference on Extending Database Technology*, pp. 136–151. – Vienna, Austria, Springer-Verlag, mars 1992.

[BFW97] Buneman (P.), Fan (W.) et Weinstein (S.). – *Path Constraints in the Presence of Types.* – Rapport technique n° MS-CIS-97-16, University of Pennsylvania, 1997.

[BFW98a] Buneman (P.), Fan (W.) et Weinstein (S.). – *Interaction between Path and Type Constraints.* – Rapport technique n° MS-CIS-98-16, University of Pennsylvania, 1998.

[BFW98b] Buneman (P.), Fan (W.) et Weinstein (S.). – Path constraints in semistructured and structured databases. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 129–138. – Seattle, Washington, juin 1998.

[BHP92] Bright (M. W.), Hurson (A. R.) et Pakzad (S. H.). – A taxonomy and curent issues in multidatabase systems. *IEEE Computer*, vol. 25, n° 3, mars 1992, pp. 50–60.

[Bid91] Bidoit (N.). – *Bases de Données Déductives (Présentation de Datalog).* – Paris, Armand Colin, 1991.

[BKKK87] Banerjee (J.), Kim (W.), Kim (H.-J.) et Korth (H. F.). – Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 311–322. – San Francisco, California, mai 1987.

[BM99] Beeri (C.) et Milo (T.). – Schemas for Integration and Translation of Structured and Semi-Structured Data. *In : Proceedings of International Conference on Database Theory (ICDT).* – Jerusalem, Israel, janvier 1999.

[BMT96] Beeri (C.), Milo (T.) et Ta-Shma (P.). – On genericity and parametricity. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 104–116. – Montreal, Canada, juin 1996.

[BNR+87] Beeri (C.), Naqvi (S.), Ramakrishnan (R.), Shmueli (O.) et Tsur (S.). – Sets and negation in a logic database language (ldl1). *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 21–37. – San Diego, California, mars 1987.

[Bor95] Borgida (A.). – Description Logics in Data Management. *IEEE Transactions on Data and Knowledge Engineering*, vol. 7, n° 5, octobre 1995, pp. 671–682.

[BPSM98] Bray (T.), Paoli (J.) et Sperberg-McQueen (C. M.). – Extensible markup language (xml) 1.0. – W3C Recommendation, février 1998. `http://www.w3.org/TR/REC-xml/`.

[Bre90] Breitbart (Y.). – Multidatabase interoperability. *SIGMOD Record*, vol. 19, n° 3, septembre 1990, pp. 53–60.

[BTBN91] Breazu-Tannen (V.), Buneman (P.) et Naqvi (S.). – Structural recursion as a query language. *In : Proceedings of International Workshop on Database Programming Languages*. pp. 9–19. – Nafplion, Greece, 1991.

[Bun97] Buneman (P.). – Semistructured data. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 117–121. – Tucson, Arizona, mai 1997. Invited Tutorial.

[CACS94] Christophides (V.), Abiteboul (S.), Cluet (S.) et Scholl (M.). – From structured documents to novel query facilities. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 313–324. – Minneapolis, Minnesota, mai 1994.

[Cat97] Cattell (R. G.). – *The Object Database Standard: ODMG 2.0*. – Morgan Kaufmann, 1997.

[CCM96] Christophides (V.), Cluet (S.) et Moerkotte (G.). – Evaluating queries with generalized path expressions. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 413–422. – Montreal, Canada, juin 1996.

[CCMS98] Christophides (V.), Cluet (S.), Moerkotte (G.) et Siméon (J.). – Optimizing Generalized Path Expressions Using Full Text Indexes. *Networking and Information Systems Journal*, vol. 1, n° 2-3, 1998. – Également présenté aux 13es Journées Bases de Données Avancées, 1997, Grenoble, France.

[CCS98] Christophides (V.), Cluet (S.) et Siméon (J.). – Semistructured and Structured Integration Reconciled – YAT += Efficient Query Processing. – novembre 1998. soumis à publication.

[CDSS98] Cluet (S.), Delobel (C.), Siméon (J.) et Smaga (K.). – Your Mediators Need Data Conversion! *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 177–188. – Seattle, Washington, juin 1998.

[CDY95] Chaudhuri (S.), Dayal (U.) et Yan (T. W.). – Join Queries with External Text Sources: Execution and Optimization Techniques. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 410–422. – San Jose, California, mai 1995.

[CH80] Chandra (A. K.) et Harel (D.). – Computable queries for relational data bases. *Journal of Computer and System Sciences*, vol. 21, n° 2, 1980, pp. 156–178.

[CH95] Chang (T.-P.) et Hull (R.). – Using Witness Generators to Support Bi-directional Update Between Object-Based Databases. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 196–207. – San Jose, California, mai 1995.

[Cha98] Chan (D. K. C.). – A Document-driven Approach to Database Report Generation. *In : International Workshop on Database and Expert Systems Applications.* pp. 925–930. – Vienna, Austria, août 1998.

[Chr96] Christophides (V.). – *Document Structurés et Bases de Données Objet.* – Thèse de doctorat, Conservatoire National des Arts et Métiers, octobre 1996.

[CHS+95] Carey (M. J.), Haas (L. M.), Schwarz (P. M.), Arya (M.), Cody (W. F.), Fagin (R.), Flickner (M.), Luniewski (A. W.), Niblack (W.), Petkovic (D.), Thomas (J.), Williams (J. H.) et Wimmers (E. L.). – Towards heterogeneous multimedia information systems: The garlic approach. *In : Research Issues in Data Engineering*, pp. 124–131. – Los Alamitos, California, mars 1995.

[CL94] Cori (R.) et Lascar (D.). – *Logique Mathematique.* – Masson, 1994.

[Clu91] Cluet (S.). – *Langages et Optimisation de Requêtes pour SGBDOO.* – Thèse de doctorat, Université Paris XI, juin 1991.

[Clu97] Cluet (S.). – Modeling and Querying Semi-Structured Data. *In : International Summer School, SCIE-97*, pp. 192–213. – Springer Verlag, 1997.

[Clu98] Cluet (S.). – Designing OQL: Allowing Objects to be Queried. *Information Systems*, vol. 23, n° 5, 1998, pp. 279–305.

[CM89] Consens (M. P.) et Mendelzon (A. O.). – Expressing Structural Hypertext Queries in GraphLog. *In : Proceedings of ACM conference on Hypertext*, pp. 269–292. – Pittsburgh, Pennsylvania, novembre 1989.

[CM93] Cluet (S.) et Moerkotte (G.). – Nested Queries in Object Bases. *In : Proceedings of International Workshop on Database Programming Languages*, pp. 226–242. – New York City, USA, août 1993.

[CM96] Cluet (S.) et Moerkotte (G.). – Query Processing in the Schemaless and Semi-structured Context. – 1996. non publié.

[Cod70] Codd (E. F.). – A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, vol. 13, n° 6, juin 1970, pp. 377–387.

[Com] Quelques systèmes d'intégration commerciaux.
`http://www.software.ibm.com/data/datajoiner/`,
`http://www.sybase.com/products/entcon/`,
`http://www.oracle.com/products/servers/rdb/html/fs_dbi.html`,
`http://www.unisql.com/product_info/unisqlm.html`,

http://www.cincom.com/totalframework/index.html,
http://www.enterworks.com/products.html,
http://www.cerfnet.com/~margie/dii/,
http://www.junglee.com/,
http://www.crossaccess.com/product.htm,
http://www.ibi.com/.

[COR98] The common object request broker: Architecture and specification. – OMG, février 1998. CORBA/IIOP 2.2.

[CS97] Cluet (S.) et Siméon (J.). – *Data integration based on data conversion and restructuring.* – Rapport technique, Verso group, octobre 1997. http://www-rocq.inria.fr/~simeon/YAT/.

[CW89] Chen (W.) et Warren (D. S.). – C-Logic of Complex Objects. *In : Proceedings of ACM Symposium on Principles of Database Systems*, pp. 369–378. – Philadelphia, Pennsylvania, mars 1989.

[DFF+98] Deutsch (A.), Fernandez (Mary), Florescu (D.), Levy (A. Y.) et Suciu (D.). – XML-QL: A Query Language for XML. – Submission to the World Wide Web Consortium, août 1998.
http://www.w3.org/TR/NOTE-xml-ql/.

[DK96] Davidson (S. B.) et Kosky (A.). – *Effecting Database Transformations Using Morphase.* – Rapport technique n° MS-CIS-96-05, University of Pennsylvania, 1996.

[DK97] Davidson (S. B.) et Kosky (A. S.). – Wol: A language for database transformations and constraints. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pp. 55–65. – Birmingham, UK, avril 1997.

[Dra94] Drakos (N.). – *The LaTeX2HTML Translator.* – Rapport technique, Computer Based Learning Unit, University of Leeds, janvier 1994.

[DST95] Delobel (C.), Souza dos Santos (C.) et Tallot (D.). – Object views of relations. *In : Congrès INFORSID.* – Grenoble, France, mai 1995.

[FFK+97] Fernandez (M.), Florescu (D.), Kang (J.), Levy (A. Y.) et Suciu (D.). – System demonstration - strudel: A web-site management system. *In : Proceedings of ACM SIGMOD Conference on Management of Data.* – Tucson, Arizona, mai 1997. Exhibition Program.

[FFK+98] Fernandez (M.), Florescu (D.), Kang (J.), Levy (A. Y.) et Suciu (D.). – Catching the Boat with Strudel: Experiences with a Web-Site Management System. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 414–425. – Seattle, Washington, juin 1998.

[FFLS97] Fernandez (M. F.), Florescu (D.), Levy (A. Y.) et Suciu (D.). – A query language for a web-site management system. *SIGMOD Record*, vol. 26, n° 3, septembre 1997, pp. 4–11.

[FFLS98] Fernandez (M.), Florescu (D.), Levy (A. Y.) et Suciu (S.). – Warehousing and incremental evaluation for web site management. *In : Proceedings of 14es Journées Bases de Données Avancées.* – Hammamet, Tunisie, octobre 1998.

[FGM+97] Fielding (R.), Gettys (J.), Mogul (J.), Frystyk (H.) et Berners-Lee (T.). – Hypertext transfer protocol – http/1.1. – RFC 2068, IETF HTTP Working Group, janvier 1997.
`http://www.w3.org/Protocols/rfc2068/rfc2068`.

[FM95] Fegaras (L.) et Maier (D.). – Towards an effective calculus for object query languages. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 47–58. – San Jose, California, mai 1995.

[FPS97] Fernandez (M.), Popa (L.) et Suciu (D.). – A Structure-Based Approach to Querying Semi-Structured Data. *In : Proceedings of International Workshop on Database Programming Languages.* pp. 136–159. – Estes Park, Colorado, août 1997.

[FS98] Fernandez (M.) et Suciu (D.). – Optimizing regular path expressions using graph schemas. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE).* – Orlando, Florida, février 1998.

[GBCG97] Guerrini (G.), Bertino (E.), Catania (B.) et Garcia-Molina (J.). – A formal view of object-oriented database systems. *Theory and Practice of Object Sytems*, vol. 3, n° 3, 1997, pp. 157–183.

[GGF+95] Gardarin (G.), Gannouni (S.), Finance (B.), Fankhauser (P.), Klas (W.), Pastre (D.), Legoff (R.) et Ramfos (A.). – IRO-DB : A Distributed System Federating Object and Relational Databases. *In : Object Oriented Multibase Systems : A Solution for Advanced Applications.* – Prentice Hall, 1995.

[Gin66] Ginsburg (S.). – *The Mathematical Theory of Context-Free Languages.* – McGraw-Hill, 1966.

[GKD97] Genesereth (M.), Keller (A.) et Duschka (O.). – Infomaster: An Information Integration System. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 539–542. – Tucson, Arizona, mai 1997. Exhibition Program.

[GLWZ98] Garcia-Molina (H.), Labio (W.), Wiener (J. L.) et Zhuge (Y.). – Distributed and parallel computing issues in data warehousing. *In : Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, p. 7. – Puerto Vallarta, Mexico, juin 1998.

[Gra93] Graefe (G.). – Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, vol. 25, n° 2, juin 1993, pp. 73–170.

[Gru90] Grumbach (S.). – *Ensembles et fonctions en Datalog.* – Thèse de doctorat, Université Paris XI, janvier 1990.

[Gun96] Gundavaram (S.). – *CGI Programming on the World Wide Web.* – O'Reilly & Associates, avril 1996.

[GW97] Goldman (R.) et Widom (J.). – Data guides: Enabling query formulation and optimization in semistructured databases. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 436–445. – Athens, Greece, août 1997.

[Ham94] Hammer (J.). – *Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems.* – Thèse de PhD, Universisty of Southern California, mai 1994.

[Har94] Hardwick (M.). – *Towards Integrated Product Databases Using Views.* – Rapport technique n° 94003, Design & Manufacturing Institute, Rensselaer Polytechnic Institute, 1994.

[Har98] Haridi (S.). – *Tutorial of Oz 2.* – Swedish Institute on Computer Science, février 1998, dfki oz documentation series édition.

[Hei95] Heiler (S.). – Semantic interoperability. *ACM Computing Surveys*, vol. 27, n° 2, juin 1995, pp. 271–273.

[HGN+97] Hammer (J.), Garcia-Molina (H.), Nestorov (S.), Yerneni (R.), Breuning (M.) et Vassalos (V.). – Template-based wrappers in the tsimmis system. *In : Workshop on Management of Semistructured Data In conjunction with PODS/SIGMOD*, pp. 532–535. – Tucson, Arizona, mai 1997.

[HKWY97] Haas (L. M.), Kossmann (D.), Wimmers (E. L.) et Yang (J.). – Optimizing queries across diverse data sources. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 276–285. – Athens, Greece, août 1997.

[Huy97] Huyn (N.). – Multiple-view self-maintenance in data warehousing environments. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 26–35. – Athens, Greece, août 1997.

[HY90] Hull (R.) et Yoshikawa (M.). – ILOG: Declarative Creation and Manipulation of Object Identifiers. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 455–468. – Brisbane, Australia, août 1990.

[HZ96] Hull (R.) et Zhou (G.). – A framework for supporting data integration using the materialized and virtual approaches. *In : Proceedings of ACM SIGMOD Conference on Management of Data.* – Montreal, Canada, juin 1996.

[ISO86] ISO. – Information processing – text and office systems – standard generalized markup language (sgml), 1986. ISO 8879:1986(E), ISO/IEC 10744.

[ISO92] *Information Technology – Hypermedia/Time-based Structuring Language (HyTime).* – 1992. ISO/IEC 10774.

[ISO94a] The express language reference manual, 1994. ISO 10303, part 11.

[ISO94b] Step data access interface (sdai), 1994. ISO 10303, part 22.

[ISO96] *DSSSL - Document Style Semantics and Specification Language.* – 1996. ISO/IEC 10179:1996.

[KK95] Kim (W.) et Kelley (W.). – On view support in object-oriented databases systems. *In : Modern Database Systems*, pp. 108–129. – Addison-Wesley, 1995.

[KKS92] Kifer (M.), Kim (W.) et Sagiv (Y.). – Querying object-oriented databases. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 393–402. – San Diego, California, juin 1992.

[KL89] Kifer (M.) et Lausen (G.). – F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 134–146. – Portland, Oregon, juin 1989.

[KLW95] Kifer (M.), Lausen (G.) et Wu (J.). – Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, vol. 42, n° 4, juillet 1995, pp. 741–843.

[KM94] Kemper (A.) et Moerkotte (G.). – Query optimization in object bases: Exploiting relational techniques. *In : Proceedings of Dagstuhl Workshop on Query Processing in Object-Oriented, Complex-Object and Nested Relation Databases*, éd. par J. C. Freytag (G. Vossen) et Maier (D.), pp. 63–98. – Morgan Kaufmann, 1994.

[Kos96] Kosky (A. S.). – *Transforming Databases with Recursive Data Structures.* – Thèse de PhD, University of Pennsylvania, 1996.

[KR96] Kuno (H. A.) et Rundensteiner (E. A.). – The multiview oodb view system: Design and implementation. *Theory and Practice of Object Sytems*, vol. 2, n° 1, 1996, pp. 202–225.

[KS91] Kim (W.) et Seo (J.). – Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, vol. 24, n° 12, décembre 1991, pp. 12–18.

[KS95] Konopnicki (D.) et Shmueli (O.). – W3qs: A query system for the world-wide web. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 54–65. – Zurich, Switzerland, septembre 1995.

[KS98] Konopnicki (D.) et Shmueli (O.). – Www information gathering : The w3ql query language and the w3qs system. *ACM Transactions on Database Systems*, 1998.

[KTV97] Kapitskaia (O.), Tomasic (A.) et Valduriez (P.). – Dealing with discrepancies in wrapper functionality. *In : Actes des 13es Journées Bases de Données Avancées (BDA'97)*, pp. 327–349. – Grenoble, France, septembre 1997.

[KYY82] Kambayashi (Y.), Yoshikawa (M.) et Yajim (S.). – Query processing for distributed databases using generalized semi-joins. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 151–160. – Orlando, Florida, juin 1982.

[LDB97] Lacroix (Z.), Delobel (C.) et Brèche (P.). – Object Views and Database Restructuring. *In : Proceedings of International Workshop on Database Programming Languages*. pp. 180–201. – Estes Park, Colorado, août 1997.

[Ler98] Leroy (X.). – *The Objective Caml system, release 2.01, Documentation and user's manual.* – Institut National de Recherche en Informatique et en Automatique,

décembre 1998.
`http://caml.inria.fr/`.

[LGr98] The lgrind Package. – Various Artists, mai 1998.
`http://www.loria.fr/services/tex/outils/lgrind.dvi`.

[LH94] Le Moal (J.-C.) et Hidoine (B.) (édité par). – *Le traitement électronique du document.* – Aix-en-Provence, France, ADBS Editions, octobre 1994. Cours INRIA.

[Lib94] Libkin (L.). – *Aspects of Partial Information in Databases.* – Thèse de PhD, University of Pennsylvania, 1994.

[LMH+85] Lohman (G. M.), Mohan (C.), Haas (L. M.), Daniels (D.), Lindsay (B. G.), Selinger (P. G.) et Wilms (P. F.). – Query processing in r*. *In : Query Processing in Database Systems*, pp. 31–47. – Springer, 1985.

[LMR90] Litwin (W.), Mark (L.) et Roussopoulos (N.). – Interoperability of multiple autonomous databases. *ACM Computing Surveys*, vol. 22, n° 3, septembre 1990, pp. 267–293.

[LMR95] Le Maitre (J.), Murisasco (E.) et Rolbert (M.). – Sgmlql, un langage d'interrogation de documents structurés. *In : Actes des 11es Journées Bases de Données Avancées (BDA'95).* – Nancy, France, août 1995.

[LMW96] Libkin (L.), Machlin (R.) et Wong (L.). – A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 229–239. – Montreal, Canada, juin 1996.

[Loh88] Lohman (G. M.). – Grammar-like functional rules for representing query optimization alternatives. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 18–27. – Chicago, Illinois, juin 1988.

[LPL96] Liu (L.), Pu (C.) et Lee (Y.). – An adaptive approach to query mediation across heterogeneous information sources. *In : Proceedings of International Conference on on Cooperative Information Systems (CoopIS)*, pp. 144–156. – Brussels, Belgium, juin 1996.

[LR82] Landers (T. A.) et Rosenberg (R.). – An Overview of MULTIBASE. *In : Proceedings of the Second International Symposium on Distributed Data Bases*, pp. 153–184. – Berlin, septembre 1982.

[LR96] Lang (B.) et Rouaix (F.). – The V6 Engine. – Proceedings the Worshop "Programming the Web", 5th International World Wide Web Conference, May 1996.

[LRO96] Levy (A. Y.), Rajaraman (A.) et Ordille (J. J.). – Querying heterogeneous information sources using source descriptions. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 251–262. – Bombay, India, septembre 1996.

[LSS96a] Lakshmanan (L. V. S.), Sadri (F.) et Subramanian (I. N.). – A Declarative Language for Querying and Restructuring the WEB. *In : Proceedings of Sixth International Workshop on Research Issues in Data Engineering*, pp. 12–21. – New Orleans, Louisiana, février 1996.

[LSS96b] Lakshmanan (L. V. S.), Sadri (F.) et Subramanian (I. N.). – SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 239–250. – Bombay, India, septembre 1996.

[Mai86] Maier (D.). – A logic for objects. *In : Workshop on Foundations of Deductive Databases and Logic Programming*, pp. 6–26. – Washington, D.C., août 1986.

[Mar92] Maranget (L.). – *La stratégie paresseuse.* – Thèse de doctorat, Université Paris VII, 1992.

[Mar98] Maranget (L.). – *HEVEA User Documentation.* – INRIA, novembre 1998. Version 1.02.

[MBE95] Mullen (J.), Bukhres (O.) et Elmagarmid (A.). – InterBase*: A Multidatabase System. *In : Object Oriented Multibase Systems : A Solution for Advanced Applications.* – Prentice Hall, 1995.

[Mel88] Melton (J.). – *ISO-ANSI SQL2.* – International Standard Organization and American National Standards Institute, 1988.

[MIKS99] Mena (E.), Illarramendi (A.), Kashyap (V.) et Sheth (A.). – OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. *Distributed and Parallel Databases*, 1999. – to appear.

[MMM97] Mendelzon (A.), Mihaila (G.) et Milo (T.). – Querying the World Wide Web. *International Journal on Digital Libraries*, vol. 1, n° 1, 1997, pp. 54–67.

[Mog96] Mogilevsky (P.). – *Integration and Translation of Heterogeneous Data.* – Thèse, Department of Computer Science, School of Mathematical Sciences, Tel-Aviv University, décembre 1996.

[MS93] Melton (J.) et Simon (A.). – *Understanding the New SQL: A complete Guide.* – Morgan Kaufmann, 1993.

[MS97] Marathe (A. P.) et Salem (K.). – A language for manipulating arrays. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 46–55. – Athens, Greece, août 1997.

[MS99] Milo (T.) et Suciu (D.). – Index Structures for Path Expressions. *In : Proceedings of International Conference on Database Theory (ICDT).* – Jerusalem, Israel, janvier 1999.

[MZ98a] Milo (T.) et Zohar (S.). – Schema Based Data Translation. *In : International Workshop on the Web and Databases (WebDB'98).* – Valencia, Spain, mars 1998.

[MZ98b] Milo (T.) et Zohar (S.). – Using Schema Matching to Simplify Heterogeneous Data Translation. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 122–133. – New York, août 1998.

[NAM97] Nestorov (S.), Abiteboul (S.) et Motwani (R.). – Inferring structure in semistructured data. *In : Workshop on Management of Semistructured Data In conjunction with PODS/SIGMOD*. – Tucson, Arizona, mai 1997.

[NAM98] Nestorov (S.), Abiteboul (S.) et Motwani (R.). – Extracting schema from semistructured data. *In : Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 295–306. – Seattle, Washington, juin 1998.

[NGT98] Naacke (H.), Gardarin (G.) et Tomasic (A.). – Leveraging mediator cost models with heterogeneous data sources. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pp. 351–360. – Orlando, Florida, février 1998.

[NUWC97] Nestorov (S.), Ullman (J. D.), Wiener (J. L.) et Chawathe (S.). – Representative objects: Concise representations of semistructured, hierarchical data. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pp. 79–90. – Birmingham, UK, avril 1997.

[O2D96] O_2 Technology, Versailles. – *The O_2DBaccess User Manual version 4.6*, septembre 1996.

[O2W96] O_2 Technology, Versailles. – *The O_2Web Reference Manual version 4.6*, septembre 1996.

[ODB94] Microsoft corp. – *Open DataBase Connectivity software development kit manual*, 1994. version 2.0.

[OPA95] Opal - integrated information and process management in manufacturing engineering, octobre 1995. Technical Annex.

[OV91] Özsu (M. T.) et Valduriez (P.). – *Principles of Distributed Database Systems*. – Prentice Hall, 1991.

[Owe93] Owen (J.). – *STEP: An Introduction*. – UK, Information Geometers Ltd, 1993.

[PAG96] Papakonstantinou (Y.), Abiteboul (S.) et Garcia-Molina (H.). – Object fusion in mediator systems. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 413–424. – Bombay, India, septembre 1996.

[Pfe95] Pfeifer (U.). – *freeWAIS-sf*. – University of Dortmund, octobre 1995, 0.5 édition.

[PGGU95] Papakonstantinou (Y.), Gupta (A.), Garcia-Molina (H.) et Ullman (J. D.). – A query translation scheme for rapid implementation of wrappers. *In : Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 97–107. – Singapore, Springer-Verlag, décembre 1995.

[PGU96] Papakonstantinou (Y.), Garcia-Molina (H.) et Ullman (J. D.). – Medmaker: A mediation system based on declarative specifications. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pp. 132–141. – New Orleans, Louisiana, février 1996.

[PGW95] Papakonstantinou (Y.), Garcia-Molina (H.) et Widom (J.). – Object exchange across heterogeneous information sources. *In : Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pp. 251–260. – Taipei, Taiwan, mars 1995.

[Pot98] Pottier (F.). – *Synthèse de types en présence de sous-typage: de la théorie à la pratique.* – Thèse de doctorat, Université Paris VII, juillet 1998.

[PR85] Postel (J.) et Reynolds (J.). – File transfer protocol (ftp). – RFC 959, ISI, octobre 1985.

[PV98a] Papakonstantinou (Y.) et Vianu (V.). – Type inference for views of semistructured data. – octobre 1998. soumis à publication.

[PV98b] Paradis (F.) et Vercoustre (A.-M.). – A language for publishing virtual documents on the web. *In : International Workshop on the Web and Databases (WebDB'98).* – Valencia, Spain, mars 1998.

[QRS+95] Quass (D.), Rajaraman (A.), Sagiv (Y.), Ullman (J. D.) et Widom (J.). – Querying semistructured heterogeneous information. *In : Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 319–344. – Singapore, Springer-Verlag, décembre 1995.

[Que90] Queinnec (C.). – *Le filtrage : une application de (et pour) Lisp.* – Paris (France), InterÉditions, 1990.

[Rag97] Raggett (D.). – HTML 3.2 Reference Specification. – W3C Recommendation, janvier 1997.
`http://www.w3c.org/TR/REC-html32/`.

[Rin98] Rinderknecht (C.). – *Une Formalisation d'ASN.1.* – Thèse de doctorat, Université Paris VI, septembre 1998.

[RLJ98] Raggett (D.), Le Hors (A.) et Jacobs (I.). – HTML 4.0 Specification. – W3C Recommendation, avril 1998.
`http://www.w3c.org/TR/REC-html40/`.

[Rou96] Rouaix (F.). – A Web navigator with applets in Caml. *In : Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking.* pp. 1365–1371. – Elsevier.

[RS97a] Roth (M. T.) et Schwarz (P. M.). – Don't scrap it, wrap it! a wrapper architecture for legacy data sources. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 266–275. – Athens, Greece, août 1997.

[RS97b] Rozenberg (G.) et Salomaa (A.). – *Handbook of Formal Languages.* – Springer-Verlag, 1997.

[Run92] Rundensteiner (E.). – Multiview: A methodology for supporting multiple views in object-oriented databases. *In : Proceedings of International Conference on Very Large Databases (VLDB)*, pp. 187–198. – Vancouver, Canada, août 1992.

[SAB+95] Subrahmanian (V. S.), Adali (S.), Brink (A.), Lu (J. J.), Rajput (A.), Rogers (T. J.), Ross (R.) et Ward (C.). – Hermes: A heterogeneous reasoning and mediator system. – 1995. Unpublished, University of Maryland.

[SAD94] Souza dos Santos (C.), Abiteboul (S.) et Delobel (C.). – Virtual schemas and bases. *In : Proceedings of the International Conference on Extending Database Technology.* – Cambridge, mars 1994.

[SC98] Siméon (J.) et Cluet (S.). – Using YAT to Build a Web Server. *In : International Workshop on the Web and Databases (WebDB'98).* – Valencia, Spain, mars 1998.

[SL90] Sheth (A.) et Larson (J.). – Federated database systems for managing distributed, heterogeneous, and autonomous databases. *Computing Surveys*, vol. 22, n° 3, 1990.

[SLT91] Scholl (M. H.), Laasch (C.) et Tresch (M.). – Updatable views in object-oriented databases. *In : Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 189–207. – Springer-Verlag, 1991.

[Sma97] Smaga (K.). – *Interface Graphique pour la Traduction de Données.* – Thèse, Université Paris VI-CNAM-ENST, septembre 1997.

[Smo92] Smolka (G.). – Feature constraint logics for unification grammars. *Journal of Logic Programming*, vol. 12, n° 1&2, janvier 1992, pp. 51–87.

[Smo95] Smolka (G.). – The Oz programming model. *In : Computer Science Today*, éd. par van Leeuwen (Jan), pp. 324–343. – Berlin, Springer-Verlag, 1995.

[Sou95] Souza dos Santos (C.). – *Un Mécanisme de Vues pour les Systèmes de Gestion de Base de Données Objet.* – Thèse de PhD, Université de Paris XI, novembre 1995.

[ST94] Smolka (G.) et Treinen (R.). – Records for logic programming. *Journal of Logic Programming*, vol. 18, n° 3, avril 1994, pp. 229–258.

[Ste94] Stephen (G. A.). – *String Searching Algorithms.* – Singapore, Lecture Notes Series on Computing - Vol. 3, World Scientific Publishing Co., 1994.

[Suc98] Suciu (D.). – Semistructured Data and XML. *In : International Conference on Foundations of Data Organization (FODO).* – Kobe, Japan, novembre 1998.

[SZ89] Shaw (G. M.) et Zdonik (S. B.). – An Object-Oriented Query Algebra. *In : Proceedings of International Workshop on Database Programming Languages*, éd. par Kaufmann (Morgan), pp. 103–112. – Gleneden Beach, Oregon, juin 1989.

[TD92] Thierry-Mieg (J.) et Durbin (R.). – *Syntactic Definitions for the ACEDB Data Base Manager.* – Rapport technique, Cambridge, CB2 2QH, UK, MRC Laboratory for Molecular Biology, 1992.

[TGS96] Tang (Z.-H.), Gardarin (G.) et Smahi (V.). – Optimizing path expressions using navigational algebraic operators. *In : Database and Expert Systems Applications, 7th International Conference, DEXA '96*, pp. 574–583. – Zurich, Switzerland, septembre 1996.

[TRV96] Tomasic (A.), Raschid (L.) et Valduriez (P.). – Scaling heterogeneous databases and the design of disco. *In : Proceedings of the 16th International Conference on Distributed Computing Systems*, pp. 449–457. – Hong Kong, mai 1996.

[Ull88] Ullman (J. D.). – *Database and Knowledge base systems.* – Computer Science Press, 1988.

[VDH97] Vercoustre (A.-M.), Dell'Oro (J.) et Hills (B.). – Reuse of information through virtual documents. *In : Second Australian Document Computing Symposium*, pp. 55–64. – Melbourne, Australia, avril 1997.

[VP97] Vercoustre (A.-M.) et Paradis (F.). – A Descriptive Language for Information Object Reuse through Virtual Documents. *In : International Conference on Object Oriented Information Systems*, pp. 299–311. – Brisbane, Australia, novembre 1997.

[VVAG97] Van den Bussche (J.), Van Gucht (D.), Andries (M.) et Gyssens (M.). – On the completeness of object-creating database transformation languages. *Journal of the ACM*, vol. 44, n° 2, mars 1997, pp. 272–319.

[VW98] Van den Bussche (J.) et Waller (E.). – Methods and views. *In : Proceedings of 14es Journées Bases de Données Avancées (BDA'98).*

[WGL+96] Wiener (J. L.), Gupta (H.), Labio (W. J.), Zhuge (Y.), Garcia-Molina (H.) et Widom (J.). – A system prototype for warehouse view maintenance. *In : Workshop on Materialized Views in cooperation with ACM SIGMOD*, pp. 26–33. – Montreal, Canada, juin 1996.

[Wid95] Widom (J.). – Research problems in data warehousing. *In : Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM'95)*, pp. 25–30. – Baltimore, Maryland, novembre 1995.

[Wie92] Wiederhold (G.). – Mediators in the architecture of future information systems. *IEEE Computer*, vol. 25, n° 3, mars 1992, pp. 38–49.

[YOL97] Yan (L. L.), Özsu (M. T.) et Liu (L.). – Accessing heterogeneous data through homogenization and integration mediators. *In : Proceedings of International Conference on on Cooperative Information Systems (CoopIS).* – Charleston, South Carolina, juin 1997.

[Zoh97] Zohar (S.). – *Schema Based Data Translation.* – Thèse, Tel Aviv University, décembre 1997.

Annexe A

Grammaire YATL

Nous donnons ici la syntaxe de YATL telle qu'elle est implantée dans le prototype.

```
program ::= input output skolems rules

input ::= "inputmodel" modellistpatterns "end"

output ::= "outputmodel" modellistpatterns "end"

modellistpatterns ::= modelpatterndesc
                  |   modelpatterndesc modellistpatterns

modelpatterndesc ::= "pattern" patternname "=" modelpatternvalue "end"

patternname ::= IDENT

modelpatternvalue ::= boolnode modelsinglepattern
                  |   boolnode "(" modellistpatterns ")"
                  |   labelnode modelsingleson
                  |   labelnode "(" modellistsons ")"
                  |   datanode

modellistpatterns ::= modelpatternvalue
                  |   modelpatternvalue "," modellistpatterns

modelsinglepattern ::= modelpatternvalue

modellistsons ::= modelson
              |   modelson "," modellistsons

modelsingleson ::= modelson

modelson ::= modeledge patternvalue

modeledge ::= "->"
```

```
          |   "=>"
          |   "-[" edgelabel "]>"

modeledgelabel ::= INT "," INT
               |   INT

boolnode ::= OR
         |   AND

labelnode ::= IDENT
          |   VARIABLE

datanode ::= TRUE
         |   FALSE
         |   INT
         |   FLOAT
         |   STRING
         |   VARIABLE
         |   patternref

patternref ::= '&' patternname
           |   '*' patternname

skolems ::= "skolem" listskolems "end"

listskolems ::= skolem
            |   skolem listskolems

skolem ::= skolemname "(" listskolemparams ")"
       |   skolemname "(" ")"

listskolemparams ::= skolemparam
                 |   skolemparam "," listskolemparams

skolemparam ::= "Bool"
            |   "Char"
            |   "Int"
            |   "Float"
            |   "String"
            |   "Label"
            |   "Ref"

rules ::= rule
      |   rule rules

rule ::= "rule" rulename "=" head ":-" body "end"
     |   "rule" rulename "inherit" rulename "=" head ":-" body "end"
```

```
head ::= headpattern
     |   skolemfonct ":" headpattern

skolemfonct ::= skolemname "(" fonctparams ")"
            |   skolemname "(" ")"

body ::= bodyliteral
     |   bodyliteral "," body

bodyliteral ::= bodypattern
             |   idvarname ":" bodypattern
             |   extfonct

extfonct ::= varname "is" fonctname "(" fonctparams ")"
         |   varname "is" fonctname "(" ")"

fonctparams ::= param
            |   param "," fonctparams

varname ::= VARIABLE

idvarname ::= VARIABLE

fonctname ::= IDENT

skolemname ::= IDENT

param ::= VARIABLE
      |   VALUE
```

Résumé

Ce travail est consacré à la conception et à la réalisation d'outils pour faciliter le développement d'applications devant accéder à des sources de données hétérogènes. Jusqu'à présent, deux approches se sont opposées. Les systèmes d'intégration structurés utilisent les technologies des bases de données. Ils fonctionnent sur de grandes quantités d'informations mais n'offrent pas la flexibilité nécessaire à un contexte réellement hétérogène. Les systèmes semistructurés utilisent des modèles de représentation sans schéma qui favorisent l'intégration simple et rapide de sources quelconques. L'absence de schéma est cependant un handicap important pour la bonne compréhension des données et pour l'évaluation efficace des requêtes.

Nous introduisons un système de types pour un modèle semistructuré et un langage déclaratif pour la spécification de programmes d'intégration. Outre la résolution des conflits structurels entre les sources, ce langage permet un traitement complet des collections et des références. Les informations de type servent à guider l'écriture des programmes et à vérifier leur conformité par rapport aux structures d'entrée et de sortie. L'utilisation de la généricité dans les programmes permet de conserver une grande flexibilité. Le système YAT implante l'ensemble de ces outils. Son architecture est conçue pour réduire au maximum le travail d'intégration. Il a ainsi permis la réalisation rapide de plusieurs applications.

Afin de décrire l'évaluation de programmes d'intégration, nous définissons un modèle opérationnel et une algèbre pour données semistructurées. Les équivalences vérifiées par les opérateurs de l'algèbre permettent non seulement de retrouver les techniques d'optimisation classiques, mais également d'en concevoir de nouvelles, spécifiques au contexte semistructuré. Enfin, la description précise des capacités opérationnelles des sources est utilisée pour répartir l'évaluation des requêtes de façon optimale entre les sources et le système d'intégration.

Abstract

We design novel techniques and tools to simplify the exploitation of heterogeneous data sources. So far, two main approaches have been competing. Structured integration systems take advantage of database techniques to offer efficient query evaluation over large sources. However, they lack the flexibility required in a truly heterogeneous environment. Semistructured systems rely on schema-less data models in order to provide simple and fast integration over any set of sources. Yet, the absence of a schema entails an increased complexity to interpret the data. It is also a major disadvantage during query evaluation.

We introduce a type system for the semistructured data model and we define YATL, a declarative integration language. YATL is able to resolve structural conflicts between sources and features high-level primitives for the manipulation of collections and references. Type information serves as a guide during the specification of integration programs and enables verifications procedures through type-checking. Moreover, YATL allows to preserve as much flexibility as possible by exploiting genericity properties. The YAT system is a prototype which implements these tools. Its architecture is designed to minimize the work required for the integration process. It has been used successfully to build various applications.

We propose an operational model suitable to describe the evaluation of YATL programs. It contains an algebra for semistructured data, whose equivalences allow us to recover all the major standard optimization techniques. It also opens some new optimization opportunities in the semistructured context. Finally, a precise description of source query capabilities within the operational model can be used to obtain an efficient distributed evaluation.