Intégration texte-représentation formelle pour la gestion de documents XML

Raphaël Troncy

Université Joseph Fourier

UFR Informatique &
Mathématiques Appliquées

Institut National
Polytechnique de Grenoble

ENSIMAG

IMAG

ÉCOLE DOCTORALE
MATHÉMATIQUES ET INFORMATIQUE

DEA D'INFORMATIQUE :
SYSTÈMES ET COMMUNICATIONS

Projet présenté par :

Raphaël TRONCY

Intégration texte-représentation formelle pour la gestion de documents XML

Effectué à l'INRIA Rhône-Alpes

Date : 21 juin 2000
Jury : Marie-France BRUANDET
Jean-Pierre CHEVALLET
Jérôme EUZENAT
Nicolas HALBWACHS

Remerciements

Je tiens à remercier Madame Marie-France Bruandet et Monsieur Jean-Pierre Chevallet, Professeurs à l'Université Joseph Fourier, d'avoir accepté de participer à ce jury.

Je remercie également Monsieur Nicolas Halbwachs, Professeur à l'Université Joseph Fourier et Responsable du DEA ISC, pour sa participation à ce jury.

Je remercie Monsieur Jérôme Euzenat, Chargé de Recherche à l'INRIA Rhônes-Alpes, de m'avoir accompagné tout au long de cette année. Ses précieux conseils et sa rigueur m'ont fait chaque jour prendre conscience de la réalité du travail de chercheur. Je le remercie tout particulièrement pour les nombreuses lectures et corrections qu'il a pu apporter à la rédaction de ce mémoire.

Je remercie le seul autre membre de l'équipe EXMO, Olivier Brunet, pour m'avoir initier au Go.

Je voudrais aussi remercier Cyril Perrin, à prescrire en cas de problèmes, car il sait toujours apporter des solutions.

Un grand merci enfin à ma famille et à tous mes amis pour leur soutien constant : Anne-Laure, Antonin, Ludovic, Nicolas, Sébastien, Yannis.

Croire ou ne pas croire,
cela n'a aucune importance.
Ce qui est intéressant,
c'est de se poser de plus en plus de questions.

Edmond Wells,
Encyclopédie du Savoir Relatif et Absolu, Tome IV.

Table des matières

Chapitre 1

Introduction

Contexte

Actuellement, le World Wide Web (WWW) contient d'importantes quantités d'informations couvrant tous les sujets imaginables. Le problème qui était avant de savoir si une information, même très spécifique, était disponible sur le Web, est maintenant devenu comment retrouver cette information. Par exemple :

- Imaginons que l'on veuille connaître les activités de recherche du chercheur nommé *Smith* ou *Feather*. Même les moteurs de recherche spécialisés dans la collecte des pages personnelles sont limités. Il y a ici un problème de précision.

- Les moteurs de recherche nous rendent une liste d'hyperliens et non directement la réponse à notre requête. Par conséquent, on ne peut pas effectuer de traitement automatique sur la réponse car un homme doit d'abord extraire l'information à traiter.

- Imaginons maintenant que l'on veuille connaître les activités de recherche de tout un groupe dont chaque membre à sa propre page. On voit qu'aucune requête n'est capable de manipuler des informations distribuées sur plusieurs sites et pages.

- Enfin, les moteurs de recherche ne sont pas capable de « deviner » des informations implicites. Ainsi, si deux chercheurs sont colaborateurs, que l'un mentionne l'information mais pas l'autre, et que l'on accède directement à la page de ce dernier, alors on passe à côté de cette information de colaboration.

Le Web a été conçu comme un espace d'informations avec pour but d'être utile non seulement à des communications entre humains, mais aussi avec des machines. Cependant la plupart des informations sur le Web sont destinées à une consommation humaine et il est donc très difficile pour une machine d'utiliser cette information qui n'est pas (ou très peu) structurée. C'est sur ce constat que l'idée d'un « *Web sémantique* » [Ber98] est née. Développer des langages et des outils pour exprimer l'information de façon à ce que celle-ci soit utilisable par des machines constitue certainement un des enjeux futurs du Web.

Apporter du sens intelligible et exploitable par des machines aux documents leurs permettra déjà d'utiliser l'information présente. Les moteurs de recherche actuels peuvent être classés en deux groupes :

- Ceux dont les catalogues sont construits « à la main » ; Les documents sont regroupés par catégorie, mais cette technique est très coûteuse étant donné la quantité requise de travail

par rapport à la vitesse à laquelle évolue le Web. De plus, les critères choisis pour obtenir de tels catalogues ne correspondent pas forcément aux attentes des utilisateurs.

– Ceux dont les catalogues sont construits automatiquement par une analyse « *en texte intégral* » des documents. Mais l'interprétation que nous faisons habituellement des résultats fournis par cette technique contient des non-sens dus en partie à la polysémie des mots ou parfois à la langue du document.

Finalement, aucune de ces techniques ne permet de faire des inférences pour, par exemple, découvrir que Val d'Isère est dans les Alpes. Le fait de décrire les documents de façon à ce que des machines puissent disposer de leur sémantique aidera sans aucun doute à améliorer les moteurs de recherche et à faire du Web une gigantesque base de connaissance.

Problématique

L'intelligence artificielle a une longue tradition dans le développement d'outils et de langages pour structurer la connaissance et l'information. Les langages de représentation de connaissance sont donc de bons candidats si l'on souhaite décrire le contenu de documents. Cette représentation du contenu va permettre de le manipuler pour faire de la recherche par analogie, par spécialisation, par similitude etc. Les travaux dans ce sens ont donné lieu à différents formalismes. Mais outre le choix du langage à utiliser, il faut répondre à plusieurs questions :

– Quel type de documents va-t-on essayer de formaliser ? A quel domaine appartiennent-ils et donc quels constructeurs de représentation de connaissance seront utiles ?

– Quels éléments du document va-t-on décrire ? S'il semble déjà exclu de représenter toute la connaissance contenue dans un document, comment isoler les éléments pertinents ? La réponse à cette question est en fait fortement liée à l'utilisation que l'on va en faire. On doit donc se demander à quel type de requête désire-t-on répondre.

– Enfin, comment intégrer cette formalisation de la connaissance au sein du document ?

Les réponses à ces questions constituent la base de notre étude dont le but est de représenter et d'intégrer la connaissance d'un corpus de documents et de fournir le système qui va tirer parti de cette formalisation.

Le projet ESCRIRE

ESCRIRE est une action de recherche coopérative entre trois projets de l'INRIA (ACACIA, EXMO et ORPAILLEUR). Son but consiste à comparer trois types de représentations de connaissances (graphes conceptuels, représentations de connaissances par objets et logiques de descriptions) du point de vue de la représentation du contenu de documents et de sa manipulation. Cette étude devrait d'abord permettre de mieux connaître les qualités respectives de chacun de ces formalismes, mais aussi mettre en évidence les propriétés intéressantes pour la recherche d'informations et déterminer les contextes favorables à l'exploitation de chacune de ces représentations. L'ambition, à terme, est de pouvoir réaliser de véritables serveurs de connaissances permettant la recherche et la manipulation des ressources appartenant, par exemple, à une entreprise.

Notre étude se place donc résolument dans le cadre de ce projet. Même si la comparaison entre les apports de ces différents formalismes n'est pas le cœur de notre propos, l'objectif poursuivi, lui, est identique : permettre la recherche et l'interrogation d'un site en s'appuyant sur le contenu des

documents. La réflexion méthodologique sur le passage de textes à leur représentation formelle sera plus spécifiquement appliquée à la drosophile.

L'existant

Le processus d'annotation de documents consistant à ajouter une représentation de leur contenu a déjà fait (et continue de faire) l'objet de multiples études. Toutes ont en commun la définition d'une *ontologie* du domaine relatif au corpus de documents annotés. Au sens philosophique du terme, une ontologie est définie comme la connaissance de l'être en tant qu'être, de l'être en soi. Dans le domaine informatique, c'est sans doute Gruber dans [Gru93] qui a le mieux redéfini ce concept. Selon lui, une « ontologie est la spécification d'une conceptualisation » dans un contexte de partage et de ré-utilisation de la connaissance. Ainsi, une ontologie est la description de concepts, de relations, d'objets et de contraintes définissant un domaine particulier. Si l'on peut encore longuement disserter sur ce qu'est ou n'est pas une ontologie, il semble plus intéressant de se concentrer sur l'utilité d'une ontologie. Les avis sont alors plus consensuels et la conception d'une ontologie doit avant tout permettre le partage et la ré-utilisation de la connaissance.

Les efforts du W3C (le comité de standardisation du Web) concernent dernièrement la mise en place d'un langage permettant d'annoter des documents. RDF (*Ressource Description Framework*) permet en effet de décrire les données d'un document et de leurs attacher des propriétés. Il est basé sur le méta-langage XML (*eXtensible Markup Language*) qui permet d'obtenir la structure du document. Ce dernier langage devient un passage obligé pour qui veut représenter la connaissance sur le Web. Cependant, si ce langage offre un excellent support syntaxique, tout reste à faire pour lui donner une sémantique, c'est-à-dire une manière d'interpréter les éléments syntaxiques dans un domaine particulier. C'est pourquoi, différents systèmes essaient de tirer partie de ce méta-langage et proposent à leur tour un langage permettant d'apporter du sens aux documents. Cette multitude de « nouveaux langages » ne contribuent pas à une standardisation mais témoignent de l'activité bouillonnante et des progrès à réaliser pour atteindre le « *Web sémantique* » qu'on nous promet.

Structure du document

Le document est organisé en trois parties. Le premier chapitre dresse un « état de l'art » des efforts déjà effectués pour donner du sens aux documents. Un survol de la représentation de connaissance et des différents formalismes existants est donc proposé. Suit une énumération des standards (ou futurs standards) du W3C pouvant contribuer à atteindre notre objectif. Nous présentons aussi quelques langages permettant d'interroger des documents XML afin de tirer partie des annotations que nous pourrions faire. Enfin, nous décrivons plusieurs systèmes intégrés qui proposent leur propre langage. Tous ont pour objectif l'annotation de documents pour permettre la description du contenu vis à vis d'une ontologie. L'accent est donc mis sur la simplicité et l'expressivité de ces langages ainsi que sur la puissance de l'application qui utilise ces annotations.

Le second chapitre revient sur la problématique posée par le sujet. Nous commençons par introduire rapidement le corpus de documents qui nous sert d'illustration à savoir les interactions géniques chez la drosophile. Nous essayons alors d'apporter une réponse à chacune des questions posées dans le cadre général. L'objectif est de pouvoir retrouver efficacement des documents à partir de requêtes structurées tirant partie des annotations. Nos réflexions se portent donc sur le type d'éléments du document à représenter, ainsi que sur la gestion de ces méta-informations.

Le troisième chapitre est consacré à la mise en œuvre des solutions apportées au sein du projet ESCRIRE , et plus spécifiquement de son instanciation dans le système de représentation de connaissance à objets (SRCO) TROEPS. Nous présentons donc les particularités de ce système avant de revenir sur les deux principales implémentations à savoir l'annotation et l'interrogation des documents. A chaque fois, nous décrivons les différents composants du processus mis en œuvre et nous donnons un exemple concrêt de résultat.

Chapitre 2

La représentation de connaissances appliquée au Web

Le but de notre étude étant de représenter la connaissance contenue dans des documents, il paraît naturel de vouloir d'abord étudier les divers formalismes de représentation de connaissances (RC). En effet, la RC a pour but de rendre possible la manipulation des connaissances au sein de la machine et cela de manière efficace et intelligible pour un utilisateur humain. Différentes approches de la représentation ont donné lieu à des formalismes concrets. Parmi ceux-ci, certains sont bâtis sur un formalisme logique, mais d'autres sont fondés sur une représentation structurée censée être plus proche d'une « représentation naturelle » d'un domaine.

Un langage de représentation de connaissances doit avant tout définir comment les informations sont décrites et en fournir une interprétation à la machine. Il doit aussi spécifier le type de requête qu'on peut formuler. Il doit être expressif, conçis, non ambigu et indépendant du contexte. La précision de la représentation et le nombre des méthodes d'inférences va limiter le domaine des requêtes possibles et la précision des réponses. Le World Wide Web peut finalement être vu comme une gigantesque base de connaissances. Cependant, cette base est assez différente de ce que l'on considère d'habitude :

- étant donné la taille du Web, il paraît déraisonnable de vouloir représenter toute la connaissance qu'il contient avec les systèmes de RC actuels ;

- le Web est dynamique et il faut donc tenir compte que les informations récoltées peuvent changer ou devenir obsolètes à tout instant.

Actuellement, de nombreux efforts tentent de standardiser la façon dont on pourrait décrire la sémantique de l'information contenue dans les pages Web. XML offre pour cela un bon support syntaxique. De même, différents travaux progressent sur un langage permettant d'interroger des documents ainsi annotés. Finalement, des systèmes complets permettent déjà de formaliser la connaissance contenue dans des documents et de l'utiliser.

Au début de ce chapitre, nous tâcherons de présenter une vision d'ensemble du domaine de la représentation de connaissances à travers l'étude de formalismes de représentation structurée (section 2.1). Ensuite, nous nous concentrerons sur les différentes initiatives du W3C gravitant autour du méta-langage XML, puisque celui-ci se destine à être le support physique commun et ceci, indépendamment du formalisme de représentation utilisé (section 2.2). La section suivante est dédiée aux divers langages qui permettent d'interroger des documents XML (section 2.3). Enfin, nous présenterons quelques systèmes intégrés, ainsi que les langages et outils développés

qui permettent de représenter la connaissance d'un corpus de documents. Nous verrons ainsi l'utilisation qu'en font ces sytèmes (section 2.4).

En conclusion (section 2.5), nous aurons identifié les problèmes spécifiques que la formalisation de la connaissance pose. Le chapitre suivant pourra alors être consacré aux solutions possibles de ces problèmes.

2.1 Quelques formalismes de représentation de connaissances

En toute généralité, représenter des connaissances propres à un domaine particulier consiste à décrire et à coder les entités de ce domaine de manière à ce qu'une machine puisse les manipuler afin de raisonner ou de résoudre des problèmes [NED00], [Kay97]. Cette définition met en évidence deux composantes complémentaires de la RC, à savoir l'expression et la manipulation des connaissances [Val99]. D'une part, les connaissances sont exprimées à l'aide d'un langage formel, dit de description des connaissances. Le langage est doté d'une *syntaxe*, précisant l'ensemble des expressions admissibles du langage et d'une *sémantique* qui permet de fournir un sens aux formules justifiant ainsi la validité des opérations effectuées. D'autre part, le but est de mécaniser un certain nombre de manipulations sur les connaissances exprimées. Ainsi, il sera nécessaire de modifier, compléter, inférer de nouvelles connaissances. Ces manipulations possibles sont spécifiées sous forme de mécanismes respectant la sémantique et opérant sur les éléments de la représentation.

Les travaux en RC ont donné naissance à de nombreux formalismes. Nous allons décrire les plus étudiés : les logiques terminologiques (section 2.1.1), les graphes conceptuels (section 2.1.2) et la représentation de connaissances par objets (section 2.1.3). Chacun de ces formalismes définit un ensemble de mécanismes concrets pour consulter des connaissances disponibles ou bien pour inférer de nouvelles connaissances qui ne sont pas déjà explicitement représentées. Ils seront donc examinés du point de vue de leurs capacités descriptives (syntaxe, sémantique) mais aussi inférentielles. Une présentation détaillée peut être trouvée dans [Euz99].

2.1.1 Les logiques terminologiques

Les logiques terminologiques (ou logiques de descriptions, ou encore langages basés sur les termes) constituent l'aboutissement d'une longue période de recherche sur la formalisation dans les langages de représentation de connaissances. Elles sont inspirées des langages de *frames* ou *schémas* proposés par Minsky [Min75] en 1975 et sont donc une des voies possibles d'évolution des idées d'origine, tout comme les SRC à objets présentés dans la section 2.1.3. La brève introduction présentée ici est principalement inspirée des travaux de Nebel [Neb90] et plus récemment de Napoli [Nap97].

Syntaxe d'une logique terminologique

Les entités manipulées par un langage de description sont les *individus*, les *concepts* et les *rôles*. Les individus correspondent à des entités concrètes de l'univers et sont décrites dans un langage assertionnel (A-box). Les concepts se réfèrent plutôt à des entités génériques. Les rôles décrivent des relations binaires entre individus. Les concepts et les rôles, eux, sont décrits dans un langage terminologique (T-box). Une base de connaissance comporte donc une terminologie et un ensemble d'assertions. Nous allons voir les caractéristiques de chacun de ces langages.

Langage terminologique

Au sein d'un système, le langage terminologique est destiné à la composition de termes structurés décrivant les concepts et les rôles. Les termes sont construits sur un ensemble $\mathcal{C}$ de concepts atomiques (dont le concept universel $\top$ et le concept absurde $\bot$) et un ensemble $\mathcal{R}$ de rôles atomiques. A partir de ces deux ensembles, des termes structurés sont construits à l'aide de connecteurs qui permettent de composer des descriptions de concepts ou de rôles, mais aussi de définir des restrictions sur les rôles. Les divers langages offrent des ensembles de connecteurs variables. Une terminologie consiste en un ensemble d'introductions de concepts. La description d'un concept précise les caractéristiques communes des individus qui sont ses instances. Cette description peut être une définition ($\doteq$) et dans ce cas on parle d'un concept *défini*. Dans le cas contraire, le concept est décrit ($\dot{\leq}$) et on parle de concept *primitif*. La différence entre les deux sortes de concepts est qu'une définition fournit des conditions nécessaires et suffisantes pour l'appartenance de l'individu au concept, alors qu'une description ne fournit que des conditions nécessaires [Val99].

[Neb90] donne un exemple de terminologie avec le langage $\mathcal{FLN}$ qui comprend les connecteurs `and` (conjontion de concepts), `all` (restriction de valeurs), `atleast` et `atmost` (restriction de cardinalité), les symboles $\doteq$ (introduction de définition), $\dot{\leq}$ (introduction de description) et $\dot{\oplus}$ (introduction d'exclusion) ainsi que `AnyThing` (concept maximal) et `anyrelation` (relation maximale).

```
Humain         ≐̇ Anything
Homme          ≐̇ Humain
Femme          ≐̇ Humain
Homme          ⊕̇ Femme
Ensemble       ≐̇ Anything
membre         ≐̇ anyrelation
Equipe         ≐̇ (and Ensemble (all membre Humain) (atleast 2 membre))
Petite-Equipe  ≐̇ (and Equipe (atmost 5 membre))
```

Cet exemple de terminologie met en jeux des concepts primitifs, et des concepts définis (`Equipe` et `Petite-Equipe`). Il est possible dans d'autres langages ($\mathcal{ALR}$) de donner une description à un rôle à l'aide de connecteurs appropriés (`androle`). Les rôles peuvent être primitifs ou définis à leur tour.

Langage assertionnel

Le langage assertionnel décrit la connaissance individuelle et factuelle. Il permet de formuler des assertions faisant apparaître les termes du langage terminologique. Ainsi, un individu est décrit par ses appartenances à un ou plusieurs concepts de la terminologie et par les rôles qui le lient à d'autres individus. Une base assertionnelle est constituée par des descriptions d'objets et des descriptions de rôles. Celles-ci sont construites à partir d'un ensemble de noms d'objets et en évoquant les termes du langage terminologique sous-jacent. Un exemple de base assertionnelle conforme avec la terminologie du paragraphe précédent est donnée ci-dessous :

```
(Humain MARY)
(Homme  DICK)
(Equipe TEAM)
(membre TEAM MARY)
(membre TEAM DICK)
(membre TEAM (atmost 2))
```

Cet exemple comporte six assertions impliquant les objets `MARY`, `DICK` et `TEAM`.

Il existe en outre deux contraintes supplémentaires [Euz99] :

- l'hypothèse de nom unique signifie qu'un nom d'objet désigne toujours la même entité ;
- l'hypothèse du monde ouvert signifie que tout fait n'étant pas explicitement nié est potentiellement vrai et donc, que l'on ne suppose pas que le monde soit complètement décrit ; Ainsi, il est possible que d'autres `Humain` existent.

Sémantique

La sémantique d'un langage terminologique est donnée moyennant une interprétation $\mathcal{E} = \langle \mathcal{D}, \mathcal{I} \rangle$ où $\mathcal{I}$ est une *fonction d'interprétation* vers un ensemble $\mathcal{D}$ dit *domaine d'interprétation*. La fonction $\mathcal{I}$ associe à chaque concept atomique un sous-ensemble de $\mathcal{D}$ et à chaque rôle atomique une relation binaire sur $\mathcal{D} \times \mathcal{D}$. La fonction d'interprétation est aussi étendue sur les expressions du langage de manière inductive comme nous le précise le Tableau 2.1 pour le langage $\mathcal{FLN}$.

$\mathcal{I}(\text{Anything})$	$=$	$\mathcal{D}$
$\mathcal{I}(\text{anyrelation})$	$=$	$\mathcal{D} \times \mathcal{D}$
$\mathcal{I}((\text{and } c_1 \ldots c_n))$	$=$	$\bigcap_{i \in 1 \ldots n} \mathcal{I}(c_i)$
$\mathcal{I}((\text{all } r\ c))$	$=$	$\{x \in \mathcal{D} \mid \forall y, \langle x, y \rangle \in \mathcal{I}(r) \Rightarrow y \in \mathcal{I}(c)\}$
$\mathcal{I}((\text{atleast } n\ r))$	$=$	$\{x \in \mathcal{D} \mid \|\{y \in \mathcal{D} \mid \langle x, y \rangle \in \mathcal{I}(r)\}\| \geq n\}$
$\mathcal{I}((\text{atmost } n\ r))$	$=$	$\{x \in \mathcal{D} \mid \|\{y \in \mathcal{D} \mid \langle x, y \rangle \in \mathcal{I}(r)\}\| \leq n\}$

TAB. 2.1 – *Interprétation des termes structurés du langage* $\mathcal{FLN}$

Pour une expression de la terminologie, sa satisfaction par rapport à $\mathcal{E}$, notée $\models_{\mathcal{E}}$ est établie comme suit :

$$\models_{\mathcal{E}} A \dot{\leq} C \quad \Leftrightarrow \quad \mathcal{I}(A) \subseteq \mathcal{I}(C)$$
$$\models_{\mathcal{E}} A \doteq C \quad \Leftrightarrow \quad \mathcal{I}(A) = \mathcal{I}(C)$$

Raisonnement terminologico-assertionnel

La relation de généralité est définie entre couples de termes t et t' vis à vis d'une terminologie $\mathcal{T}$. Ainsi, on dira que t' subsume t (notée $t \preceq_{\mathcal{T}} t'$) si et seulement si dans tout modèle $\mathcal{E}$ de $\mathcal{T}$, $\mathcal{E}(t) \subseteq \mathcal{E}(t')$. Le test de subsomption est à la base d'un ensemble d'autres opérations nécessaires pour le raisonnement terminologique. En particulier, la subsomption permet de vérifier l'incohérence d'un concept ($t \preceq_{\mathcal{T}}$ `Nothing`), l'équivalence ($t \approx_{\mathcal{T}} t'$) ou l'exclusion ($t \oplus_{\mathcal{T}} t'$ ou (`and` $t\ t'$) $\preceq_{\mathcal{T}}$ `Nothing`) entre deux concepts.

L'intérêt des logiques de description réside donc dans la possibilité de définir des termes structurés (concepts et rôles) et de calculer, de manière efficace, la subsomption entre ces termes. Mais cette obligation de définir tous les termes impliqués dans le test de subsomption est en règle générale impossible si l'on souhaite modéliser un domaine réaliste. En effet, il est bien difficile de trouver systématiquement des conditions nécessaires et suffisantes d'appartenance à un concept ou un rôle pour une entité. De plus, les principaux travaux portent sur la partie terminologique et peu sur la partie assertionnelle compte tenue de sa complexité. Il est à noter que les logiques

terminologiques s'adaptent bien au Web, c'est-à-dire à un monde ouvert puisque tous les faits non présents ne sont pas forcément faux.

2.1.2 Les graphes conceptuels

Les graphes conceptuels ont été introduits par Sowa en 1984 [Sow84]. Ils sont fortement inspirés des réseaux sémantiques [Qui68] qui sont fondés sur un modèle graphique permettant de combiner la représentation des concepts (sous forme de nœuds) et des relations entre concepts (sous forme d'arcs). Un mécanisme de raisonnement fondé sur le parcours de la structure graphique permettait d'établir des liens entre des concepts du même réseau. De plus, les réseaux sémantiques avaient introduit la notion d'héritage, matérialisée par un arc particulier (*sorte-de*) entre les nœuds. L'exposé présenté ici est inspiré des notes de cours de [Euz99], lui-même fondé sur le traitement formel des graphes conceptuels proposé dans [CM92] et [MC96].

Syntaxe d'un langage de graphes conceptuels

Un graphe conceptuel est un graphe connexe bipartie composé de deux types de nœuds : des *concepts* et des *relations*. Les nœuds relations possèdent un ou plusieurs arcs qui les lient aux nœuds concepts. Les concepts désignent des entités, des propriétés, des états ou des évènements. Ils consistent en une étiquette, c'est-à-dire un couple (*type de concept*, *référent*). Le référent peut être soit individuel et désigner un individu concret, soit générique (noté par $*$) et il désigne alors n'importe quel individu. Le type est un type d'individus, tiré du *treillis de types*. Chaque type définit un ensemble de contraintes sur la structure des individus qui sont des instances de ce type. Une relation lie un certain nombre de nœuds conceptuels et est d'arité fixe.

Le vocabulaire (ensemble de types, référents et relations) est défini moyennant la notion de *support* permettant de contraindre la construction des graphes. Ainsi, l'ensemble de types est organisé par la relation dite de sous-typage en un treillis de types. Ce treillis est muni d'éléments minimal et maximal. Un ordre est également spécifié entre les référents individuels qui sont deux à deux incompatibles mais plus spécifiques que le référent générique ($*$). Finalement, un ensemble de graphes-étoiles permet de fixer les signatures des relations utilisées. Un graphe étoile est composé d'un seul nœud relationnel, mais lié à tous les autres nœuds conceptuels. La signature définit les types maximaux (pour le sous-typage) pour les nœuds conceptuels qui peuvent être connectés à chaque arc. L'exemple de la Figure 2.1 illustre notre propos en donnant un exemple avec le treillis de types et les graphes étoiles.

Formellement, un support $\mathcal{S}$ est un quintuplet $\langle T_C, T_R, M, ::, B\rangle$ tel que :

- T_C est l'ensemble des types de concepts ;
- T_R est l'ensemble des types de relations ;
- M est un ensemble énumérable de marqueurs contenant $*$ (le marqueur générique) et $\emptyset$ (le marqueur absurde) ;
- $::$ est un prédicat de conformité sur $M * T_C$;
- B est un ensemble de graphes étoiles sur T_C, T_R.

et :

- $\langle T_C, \leq\rangle$ est un ordre partiel fini avec $\top$ et $\bot$ comme supremum et infimum ;
- $T_C \cap T_R = \emptyset$;

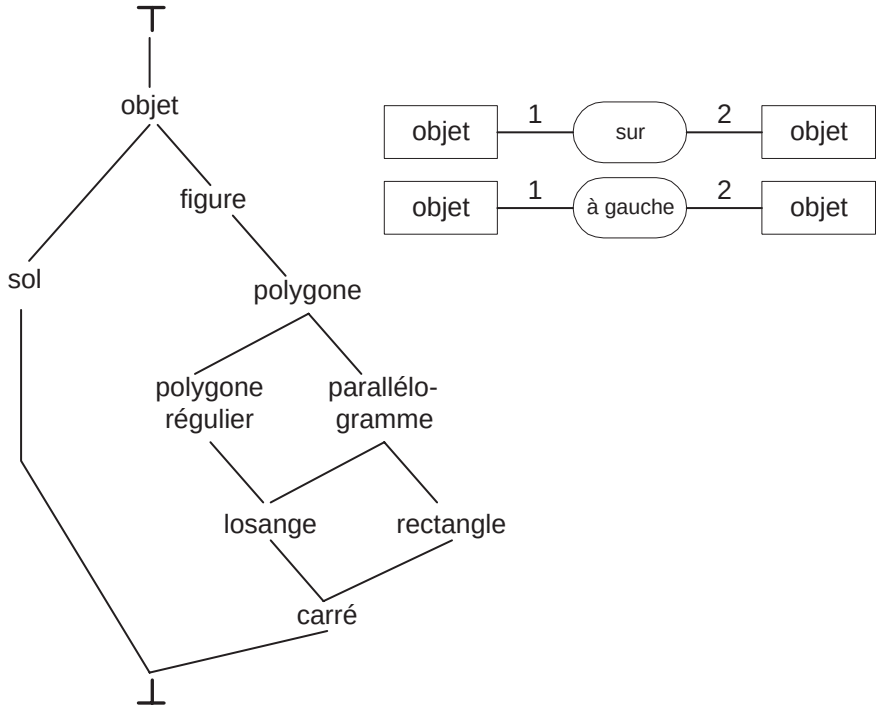

FIG. 2.1 – *Exemple de support avec le treillis de types et les graphes étoiles*

– il existe une bijection $\beta : T_R \longrightarrow B$ telle que $\forall r \in T_R, \beta(r)$ soit un graphe étoile pour r ;

– $\langle M, \sqsubseteq \rangle$ soit un treillis construit de telle sorte que $\forall m \in M - \{*, \emptyset\}, \emptyset \sqsubseteq m \sqsubseteq *$ et que tous les éléments de $M - \{*, \emptyset\}$ soient incomparables ;

– $\forall m \in M, \forall t, t' \in T_C$,

$$\begin{array}{ll} m :: \top, & (t' \leq t) \wedge (m :: t') \Rightarrow m :: t \\ m :: \bot, & (m :: t') \wedge (m :: t) \Rightarrow m :: t' \wedge t \\ \bot :: t, & t \neq \top, \equiv * :: t. \end{array}$$

Un graphe conceptuel sur le support $\mathcal{S}$ est donc un hypergraphe $G = \langle R, C, U, l \rangle$ tel que :

– $\langle R, C, U \rangle$ soit un graphe bipartie fini avec $C \neq \emptyset$;

– l soit une fonction

$$\begin{array}{rcl} l & : & R \longrightarrow T_R \\ & & r \longmapsto type(r) \\ & & C \longrightarrow T_C * (M - \{\emptyset\}). \\ & & c \longmapsto \langle type(c), ref(c) \rangle. \end{array}$$

– $\forall c \in C, ref(c) :: type(c)$;

– $\forall r \in R$,

1. les arcs sortants de r sont totalement ordonnés de 1 à $degré(r)$ (et leurs extrémités sont nommées $G_i(r)$) ;

2. $degré(r) = degré(\beta(type(r)))$;
3. $\forall i \in 1, ...degré(r), type(G_i(r)) \leq \beta_i(type(r))$.

Des types complexes peuvent ainsi être définis à l'aide des graphes conceptuels et on associe un type à un graphe. La Figure 2.2 nous donne un exemple de graphes sur le support de la Figure 2.1.

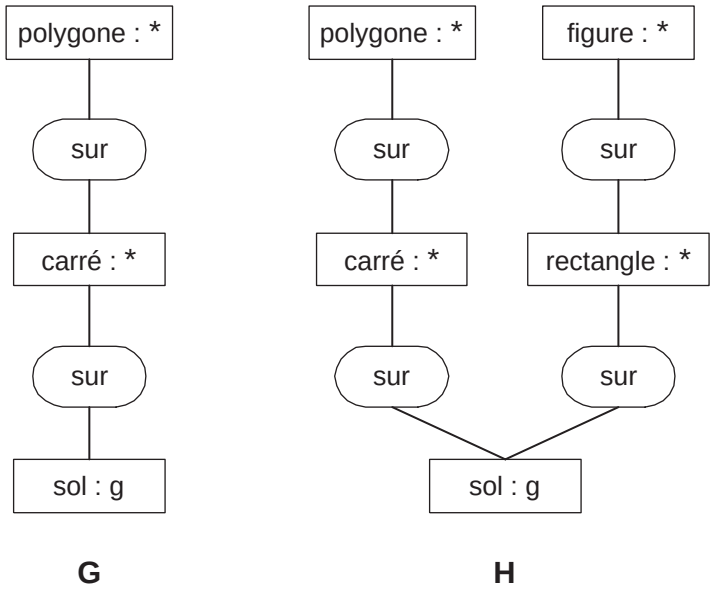

FIG. 2.2 – *Exemple de graphes sur le support de la Figure 2.1*

Sémantique

La sémantique d'un graphe conceptuel est donnée par une traduction vers le calcul des prédicats. A chacun des éléments du support, on associe donc son équivalent dans un langage logique d'ordre un. Ainsi, les référents génériques sont traduits par des variables et les référents individuels par des constantes. Chaque type, conceptuel ou relationnel, est traduit par un prédicat. Ceux correspondant aux concepts sont monadiques, alors que ceux correspondant aux relations ont une arité égale au nombre d'arcs de la relation.

Soit un support $\mathcal{S} = \langle T_C, T_R, M, :, B \rangle$, on considère un calcul des prédicats basé sur un ensemble de constantes $K \supseteq M - \{*, \emptyset\}$, un ensemble de symboles de prédicats $P \supseteq T_C \cup T_R$ et un ensemble de variables V. Les prédicats associés à T_C sont d'arité 1, alors que les prédicats associés à T_R ont une arité égale au nombre d'arcs de la relation sous-jacente. Soit donc un S-graphe $\langle R, C, U, l \rangle$, on définit :

$$\begin{array}{rcl} \iota & : & C \longrightarrow V \cup K \text{ injective sur } V \\ & & c \longmapsto \left\{ \begin{array}{l} x \in V \text{ si } ref(c) = * \\ ref(c) \in K \text{ sinon} \end{array} \right. \\ \phi & : & C \cup R \cup S\text{-graphe} \longrightarrow CP1. \end{array}$$

$$c \longmapsto type(c)(\iota(c)).$$
$$r \longmapsto type(r)(\iota(G_1(r)), ..., \iota(G_{degré(r)}(r)))$$
$$G \longmapsto \exists_{c \in C; ref(c)=*} \iota(c) \bigwedge_{c \in C} \phi(c) \wedge \bigwedge_{r \in R} \phi(r)$$

Le graphe G de la Figure 2.2 se traduit par :

$$\exists x, y; sol(g) \wedge polygone(x) \wedge carré(y) \wedge sur(x, y) \wedge sur(y, g)$$

et le graphe H par :

$$\exists x, y, z, w; polygone(x) \wedge figure(y) \wedge carré(z) \wedge rectangle(x) \wedge sol(g) \wedge sur(x, z) \wedge sur(z, g) \wedge sur(y, w) \wedge sur(w, g)$$

Raisonnement

Une relation de généralisation/spécialisation (notée $\preceq_S$) est définie entre graphes conceptuels. Ainsi, un graphe conceptuel G est plus spécifique qu'un autre H si, intuitivement, il contient toutes les informations véhiculées par H et en ajoute d'autres. Cette généralisation/spécialisation s'exprime sur les graphes par une projection qui plaque un graphe sur un autre tout en vérifiant la spécialisation des étiquettes sur les concepts et les relations et la préservation de l'ordre des arcs. Ainsi, $G \preceq_S H$ si et seulement si il existe une projection de H vers G. Formellement, une S-projection entre deux S-graphes $\langle R, C, U, l\rangle$ et $\langle R', C', U', l'\rangle$ est une paire d'applications

$$\langle f : \ R \longrightarrow R',$$
$$g : \ C \longrightarrow C'\rangle$$

telle que :

- $\forall r \in R, \forall i \in 1, ...degré(r), g(G_i(r)) = G'_i(f(r))$;
- $\forall r \in R, l'(f(r)) = l(r)$;
- $\forall c \in C, type'(g(c)) \leq type(c)$ et $ref'(g(c)) \sqsubset ref(c)$.

Ainsi, on peut obtenir G en appliquant une suite d'opérations élémentaires de *spécialisation* à H. Les opérations élémentaires de *généralisation* sont obtenues en inversant ces opérations.

Les graphes conceptuels introduisent donc une séparation entre les connaissances de nature différente. La description des types conceptuels (et de leurs relations de sous-typage) et la caractérisation des relations sont organisées dans le support, alors que les faits sont organisés au sein d'un ensemble de graphes. Mais il n'y a pas de distinction claire entre classe et instance. Les graphes conceptuels disposent d'un formalisme relationnel mais surtout, ne sont pas contraints à un sujet. En revanche, il n'y a pas (ou très peu) de définitions de types.

2.1.3 La représentation de connaissances par objets

Les langages de représentation par objets sont inspirés de la programmation par objets, mais représentent aussi une tentative de rationalisation du modèle des *frames* en distinguant notamment classes et instances [DEMN98]. Ainsi, le langage de description des connaissances s'articule autour de la notion d'*objet structuré*, de *classe* et d'*instance*. De plus, les systèmes de représentation de connaissances par objets (SRCO) offrent des mécanismes d'exploitation particuliers comme l'héritage ou la classification.

Principes de représentation

Le langage de description des connaissances distingue deux types d'unités syntaxiques : les instances représentant des individus concrets du domaine modélisé et les classes regroupant des individus par catégorie dans ce domaine. Ces deux types ont une structure similaire à savoir un ensemble d'attributs auxquels un nom est affecté. De plus, chaque objet possède un lien qui le connecte à une unité plus générale (`est-un` pour les instances et `sorte-de` pour les classes). Les classes sont organisées dans une structure hiérarchique, appelée *taxonomie*, selon leur généralité.

Les classes représentent donc des catégories d'un domaine. A l'instar des classes en LPOO, une classe a pour fonction la génération d'instances. La description d'un attribut de classe délimite les valeurs que cet attribut peut prendre au sein d'une instance de la classe. Ainsi, la description des attributs fournit aux instances les conditions nécessaires d'appartenance à une classe.

L'instance d'une classe est un objet physique proprement dit dans la mesure où elle représente une entité individuelle et identifiable du domaine. En tant que structure de données, un objet consiste en un ensemble de couples <attribut, valeur>, à laquelle un identificateur est associé. Les attributs de l'objet sont fixés par sa classe d'appartenance à laquelle il est lié par le lien `est-un`.

Un attribut est l'expression d'une caractéristique d'un individu ou d'une classe d'individus du domaine modélisé. Cette caractéristique peut être une propriété qui permet de décrire les individus indépendamment de tout autre individu, ou alors une relation qui lie l'objet à un ou plusieurs autres objets. Par conséquent, la valeur d'un attribut peut soit appartenir à un type de données simple (entier ou chaîne de caractères par exemple), soit être un objet.

Relations entre entités de description

Les relations entre les expressions d'un langage à objets sont réglementées par une sémantique dénotationnelle du langage fondée sur une interprétation ensembliste des classes.

Les objets membres d'une classe sont liés à celle-ci par une relation d'appartenance. Ainsi, l'entité représentée est incluse dans l'interprétation de la classe. Les valeurs des attributs d'une instance doivent bien sûr être compatibles avec les restrictions spécifiées sur les attributs de sa classe.

Les classes sont organisées en hiérarchie selon une relation de généralité, la *spécialisation*. Cette relation d'ordre partiel entre deux classes distingue la classe la plus spécifique de la plus générale. Le lien `sorte-de` va lier une classe à sa super-classe directe, et toutes ses instances seront aussi instances de ses super-classes. La spécialisation a donc une sémantique qui repose sur l'inclusion ensembliste. Ainsi, l'extension d'une classe est incluse dans celle de chacune de ses super-classes. La spécialisation structure l'ensemble des classes en une hiérarchie qu'on appelle parfois *graphe d'héritage*. Enfin, la description d'une sous-classe affine celle de sa super-classe en ajoutant de nouveaux attributs et/ou en restreignant le domaine des attributs déjà existants.

L'héritage, bien que souvent confondu avec la spécialisation, est en fait un mécanisme qui permet de profiter de l'organisation hiérarchique des classes en optimisant par exemple le stockage des connaissances exprimées. L'héritage peut être multiple si on autorise une classe à avoir plusieurs super-classes directes et incomparables.

Mécanismes d'exploitation

Tout comme les autres formalismes de représentation, un système à objets disposent de mécanismes permettant d'exploiter les connaissances. Les mécanismes principaux qui opèrent sont l'inférence de valeur, l'héritage ou la classification.

Les mécanismes d'inférence de valeur permettent de calculer la valeur d'un attribut quand celle-ci n'est pas spécifiée pour un objet donné. L'héritage favorise la factorisation des connaissances au sein d'une taxonomie de classes. La classification fait partie des mécanismes d'inférence les plus sophistiqués qu'un modèle à objets peut offrir. Son rôle est de compléter les connaissances en recherchant la place de l'entité à introduire (instance ou classe) dans la hiérarchie de spécialisation [Val99].

Les formalismes à objets offrent donc un mode de représentation conceptuel dans la mesure où ils combinent la représentation des connaissances concernant la description des individus du monde modélisé et celles concernant la manière d'obtenir la valeur de certaines caractéristiques. On doit finalement souligner le rôle primordial de la taxonomie de classes puisqu'elle permet à différents mécanismes de manipuler et d'exploiter les connaissances.

2.1.4 Conclusion

Nous venons d'exposer brièvement les aspects descriptifs et inférentiels de plusieurs approches vers la représentation de connaissances. Voici donc une synthèse résumant la façon dont les entités individuelles sont décrites et leur rapport avec des entités plus génériques.

Les logiques de description dénotent l'expression des connaissances individuelles par des constantes. La description est composée par l'ensemble des atomes qui font apparaître cette constante. Dans les graphes conceptuels, les individus sont dénotés par des marqueurs individuels. La description se résume alors à un graphe unique. Les SRCO offrent une expression structurée dans la mesure où l'identité et la description des entités sont regroupées en une seule expression, l'objet. Au sein de cette expression, chaque caractéristique de l'entité donne suite à un attribut. Le nombre et la nature des caractéristiques des individus en logiques de description et en RCO sont déterminés par leur appartenance à un concept / classe. De telles restrictions ne font pas partie des modèles de graphes conceptuels.

Le rapport entre les individus et les entités génériques se traduit, en logiques de description, par les concepts caractérisant leurs instances avec l'ensemble des rôles et des restrictions sur le co-domaine de ces rôles. Dans le modèle des graphes conceptuels, la distinction entre les expressions décrivant une entité individuelle et un groupe n'est pas très nette. En effet, une classe décrit ses membres par un sous-graphe qui est plus général que chacune des descriptions des membres. La classe définit donc une sous-structure commune avec des possibles variations pour chaque nœud conceptuel. L'appartenance à la classe signifie l'existence d'une projection du graphe de la classe dans le graphe de l'individu. Les connaissances génériques dans les SRCO sont organisées en classes qui définissent, dans certains cas, la structure des objets membres, mais surtout la variabilité dans les valeurs des attributs. Ainsi, les attributs des classes fournissent des domaines de valeurs, qui sont soit des types, soit des classes, pour les attributs des objets. L'instanciation qui lie les objets à leurs classes se traduit alors, sur chaque attribut, par une relation d'appartenance de sa valeur dans l'objet au domaine défini par la classe.

Ces divers formalismes se distinguent donc par la richesse et la finesse des langages de description des connaissances, par l'efficacité des mécanismes d'inférence intégrés et par leur capacité à faire évoluer la connaissance représentée.

2.2 XML et les technologies adjacentes

Le W3C (le comité de standardisation du Web) travaille depuis quelques années à l'élaboration d'un langage permettant l'échange de connaissances structurées entre des machines. Ainsi est

né XML (*eXtensible Markup Language*) dont la première recommandation [BPS98] (ou en français [ACY98]) date de février 1998. Ce méta-langage permet de définir ses propres balises pour structurer les informations contenues dans un document. Mais dans sa tentative d'accéder au sens d'une page Web, le W3C n'en est pas à son premier coup d'essai puisque déjà dans l'évolution de HTML 3 à HTML 4, des balises permettent d'indiquer le sens de certaines informations.

Nous allons donc revenir sur chacune des différentes techniques qui permettent à une machine d'accéder à la structure et à la sémantique des informations qu'elles manipulent. Nous analyserons quelques balises spécifiques de HTML ainsi que le mécanisme des feuilles de style (section 2.2.1) avant de présenter le langage XML lui-même (section 2.2.2). Nous regarderons enfin RDF (*Ressource Description Framework*), un langage issu de XML qui permet d'ajouter une repésentation de la sémantique d'un document sans faire aucune hypotèse sur sa structure (section 2.2.3).

2.2.1 La sémantique dans HTML

Les balises `<META>` et `<SPAN>` de HTML

Historiquement, la première tentative pour décrire certaines informations contenues dans une page Web s'est effectuée par l'intermédiaire de la balise `<META>` de HTML [VF99]. Cependant, cette balise permet seulement d'appliquer une propriété globale à l'ensemble du document, comme par exemple :

```
<HEAD>
      <META NAME="author" CONTENT="Raphaël">
</HEAD>
```

Cela exprime que `Raphaël` est l'auteur du document tout entier. Le mécanisme d'ancre de HTML permet toutefois d'appliquer des propriétés à des parties spécifiques de la page.

```
<HEAD>
      <META NAME="author" CONTENT="#L0">
      <META NAME="tel" CONTENT="#L1">
</HEAD>
<BODY>
      This page is written by
      <SPAN ID="L0">Raphaël Troncy</SPAN>
      His telephone number is <SPAN ID="L1">54 18</SPAN>
</BODY>
```

Ainsi, cet exemple nous montre que le contenu du type indiqué par l'attribut `NAME` de la balise `<META>` peut être trouvé à un endroit spécifié dans le document. Bien que ce ne soit pas le but originel de cette balise, on voit que ce mécanisme permet de décrire la sémantique de certaines informations. C'est d'ailleurs sur cette idée qu'est né SHOE, puisqu'il propose une extension du concept de la balise `<META>` comme nous le verrons dans la section 2.4.1.

Selon la spécification 4.0 de HTML, la balise `<SPAN>` est « un container générique pour un élément textuel offrant un mécanisme général pour renforcer la structuration d'un document » [ACC+97]. Ainsi, en utilisant l'attribut standard `CLASS`, on peut obtenir le même marquage que ci-dessus :

```
<BODY>
      This page is written by
      <SPAN CLASS="author">Raphaël Troncy</SPAN>
      His telephone number is <SPAN CLASS="tel">54 18</SPAN>
</BODY>
```

Le document de référence de HTML 4.0 suggère d'ailleurs l'utilisation de la balise `<SPAN>` pour exprimer la sémantique d'un document. ONTOBROKER est basé sur la même idée, mais il utilise l'ancre HTML `<A>` au lieu de la balise `<SPAN>` comme nous le verrons dans la section 2.4.2.

Les feuilles de style CSS

Les feuilles de style CSS ont pour but de séparer la structure d'un document de sa future mise en forme. Ainsi, les éléments du document peuvent être formattés selon un style spécifié. Mais on peut aussi utiliser (et abuser de) la balise `<STYLE>` pour ajouter des informations sémantiques. L'exemple précédent deviendra :

```
<HEAD>
      <STYLE>
            SPAN.L0 {contents:author}
            SPAN.L1 {contents:tel}
      </STYLE>
</HEAD>
<BODY>
      This page is written by
      <SPAN CLASS="L0">Raphaël Troncy</SPAN>
      His telephone number is <SPAN CLASS="L1">54 18</SPAN>
</BODY>
```

2.2.2 XML

Le langage XML se veut un langage permettant de définir la syntaxe d'un document structuré, et donc un format d'échange entre différentes applications. C'est un langage de balisage qui se présente comme un sous-ensemble de SGML. Son but est de permettre au SGML générique d'être transmis, reçu et traité sur le Web de la même manière que l'est HTML aujourd'hui. XML a été conçu pour être facile à mettre en œuvre et interopérable avec SGML et HTML. Une bonne introduction en français de ce langage se trouve en [BB99] ou en [Laz98].

Un petit exemple

Un document XML est composé de balises (ou *tag*) et de contenus. Voici un petit exemple issu de [Wal97] :

```
<?XML version="1.0"?>
<oldjoke>
      <burns>Say <quote>goodnight</quote>, Gracie.</burns>
      <allen><quote>Goodnight, Gracie.</quote></allen>
      <applause/>
</oldjoke>
```

1. Le document commence par une « instruction de traitement » (ou *processing instruction*) désignant un en-tête :

   ```
   <?xml version="1.0" encoding="ISO-8859-1" standalone="yes|no"?>
   ```

   Cette ligne indique :

   - que c'est un document XML, respectant la version 1.0 ;
   - la norme de codage des caractères (`UTF-8, UTF-16, ISO-8859-1, etc.`) ;

- si le document est autonome (`standalone="yes"`) ou pas (`standalone="no"`), c'est-à-dire s'il n'a pas de DTD, ou au contraire s'il en a une.

2. Si le document fait référence à une DTD (« *Document Type Description* ») externe, celle-ci est déclarée dans l'en-tête par :

```
<!DOCTYPE <élément_racine>  SYSTEM "<nom_fichier.dtd>">
```

Dans notre exemple, l'élément racine serait `oldjoke`.

Les différentes balises

Il existe 6 types de balise qui peuvent apparaître dans un document XML : des éléments, des commentaires, des références à une entité, des sections marquées, des « instructions de traitement », et une DTD interne.

1. L'élément est la forme de balise la plus courante. Un élément commence après une balise de début et fini par une balise de fin :

```
<élément> ... </élément>
```

Un élément peut être vide, c'est-à-dire ne pas avoir de contenu, et alors, il peut s'écrire de deux façons :

```
<élément>  </élément>      ou      <élément/>
```

Un élément peut contenir des attributs qui sont des paires attributs-valeurs. La valeur peut être indiqué entre *quote* (`'...'`) ou *double quote* (`"..."`) :

```
<élément att1="val1" att2="val2" ... attn="valn">
```

2. Des commentaires peuvent être insérés entre les éléments d'un document XML. Ils sont contenus entre les balises :

```
<!--     ...     -->
```

Un commentaire peut contenir n'importe quel caractère excepté la chaîne `"--"`.

3. Un document XML peut faire référence à des entités. Une entité est identifiée par un nom, et on y fait référence en plaçant ce nom entre les caractères spéciaux '`&`' et '`;`' :

```
&<nom_de_l'entité>;
```

Alors, le bloc correspondant à cette entité est littéralement recopié à l'endroit où l'on y fait référence. On peut ainsi utiliser un certain nombre de caractères qui étaient réservés :

```
&amp;       &gt;       &lt;       &apos;       &quot;
```

affichera respectivement les caractères : `&, >, <, ', "`.
Une dernière possibilité permet d'afficher n'importe quel caractère grâce à son code décimal ou hexa-décimal dans la norme Unicode U+211E :

`&#<code_décimal>;` ou `&#x<code_hexa-décimal>;`

Ainsi, (`&#48;`) ou (`&#x30;`) affichera le caractère `0`.

4. Une section marquée *CDATA* est une zone spéciale dans le document, où tous les caractères réservés vont être ignorés. On peut donc à l'intérieur d'une telle section écrire directement les caractères `<` ou `&` par exemple, sans qu'ils soient reconnus comme marqueurs. Une section *CDATA* se déclare comme suit :

   `<!CDATA[ ... ]]>`

   La seule chaîne de caractères interdite dans une section *CDATA* est : `"]]>"`. Ainsi, les commentaires ne seront pas reconnus comme tels, mais considérés comme une chaîne normale.

5. Les « instructions de traitement » permettent de fournir des informations à une application extérieure. Comme les commentaires, elles ne font donc pas vraiment partie du document XML. Elles ont la forme :

   `<?<nom> <informations>?>`

   Le nom identifie les IT. Les noms commençant par *xml* sont réservés à la standardisation de XML.

6. Un document XML peut enfin contenir une DTD interne, et/ou faire référence à une DTD extérieure. Dans ce cas, celle ci doit se trouver dans l'en-tête du document :

   ```
   <!DOCTYPE <élément_racine> SYSTEM "<fichier_extérieur.dtd>" [
                   <contenu_de_la_DTD_interne> ]>
   ```

   Nous allons voir maintenant en quoi consiste une DTD.

La DTD

Une des grandes forces de XML est de pouvoir créer ses propres balises. Cependant, si ces balises ne sont pas contraintes par une grammaire, aucune application ne pourra interpréter le document. Par exemple, pour pouvoir présenter un document grâce à une feuille de style, l'organisation des balises doit être spécifiée. Plus généralement, une description du document va définir un ensemble de méta-informations incluant l'ordre et l'arrangement possible des balises, les types et valeurs par défaut des attributs, les entités qui peuvent être rencontrées, et le nom de tous les fichiers externes référencés. Il existe 4 sortes de déclaration dans une DTD XML : des déclarations d'éléments, des déclarations de liste d'attributs, des déclarations d'entités, et des déclarations de notations.

1. La déclaration d'un élément identifie son nom et la nature de son contenu.

   `<!ELEMENT <nom> (<élément`$_1$`>, ... ,<élément`$_n$`>)>`

La virgule (`,`) entre les noms des éléments indique qu'ils apparaissent successivement. De plus, un marqueur peut être ajouté à la suite d'un nom d'élément :

- le point d'interrogation (`?`) signifie que l'élément peut apparaître 0 ou 1 fois ;
- le plus (`+`) signifie que l'élément va apparaître au moins 1 fois, mais peut être répété plusieurs fois ;
- l'étoile (`*`) signifie que l'élément peut apparaître 0 ou plusieurs fois ;
- l'absence de ponctuation indique que l'élément doit apparaître exactement 1 fois.

La barre verticale (`|`) s'interprète comme un OU logique. En plus de noms d'éléments, le symbole `#PCDATA` est réservé pour indiquer une donnée de type caractère (*PCDATA = parseable character data*). Les éléments qui contiennent à la fois d'autres éléments et un contenu *PCDATA* sont considérés comme ayant un contenu mixte qui doit être optionnel (`*`). Enfin, deux autres contenus sont possible :

- `EMPTY` indique que l'élément n'a pas de contenu ;
- `ANY` indique que n'importe quel contenu est permis.

L'exemple suivant constitue l'ensemble des déclarations des éléments rencontrés dans l'exemple introductif.

```
<!ELEMENT oldjoke  (burns+, allen, applause?)>
<!ELEMENT burns    (#PCDATA | quote)*>
<!ELEMENT allen    (#PCDATA | quote)*>
<!ELEMENT quote    (#PCDATA)*>
<!ELEMENT applause EMPTY>
```

2. La déclaration d'attributs identifie les éléments qui possèdent des attributs, et quels attributs ils ont. Chaque attribut dans la déclaration est constitué de trois parties : un nom, un type, et une valeur par défaut.

   `<!ATTLIST <nom_élément> ... <attribut`$_i$`> <type`$_i$`> <valeur`$_i$`> ... >`

   Il y a 6 types d'attribut possible :

   - `CDATA` : la valeur est une chaîne de caractères, où, à la différence des sections *CDATA*, les marqueurs spéciaux sont reconnus et donc les références aux entités remplacées par leurs corps ;
   - `ID` : la valeur est un nom unique ;
   - `IDREF` ou `IDREFS` : la valeur est un nom qui a déjà été déclaré comme de type `ID` dans le document ;
   - `ENTITY` ou `ENTITIES` : la valeur est l'identifiant d'une entité ;
   - `NMTOKEN` ou `NMTOKENS` : la valeur est une forme restreinte d'une chaîne de caractères ;
   - type énuméré : la valeur est prise dans la liste de noms fournis.

   Il y a 4 types de valeur par défaut possible :

   - `#REQUIRED` : pour toute occurence de l'élément dans le document, une valeur de l'attribut doit être explicitement spécifiée ;

- `#IMPLIED` : la valeur de l'attribut n'est pas requise, et aucune valeur par défaut n'est spécifiée ;
- `"<valeur>"` : l'attribut peut prendre n'importe quelle valeur, mais si celle-ci n'est pas présente, il aura la valeur `<valeur>` ;
- `#FIXED "<valeur>"` : ici, l'attribut n'est pas obligatoire, mais s'il apparaît, il aura la valeur `<valeur>` spécifiée.

3. La déclaration d'une entité permet d'associer un nom avec une partie d'un document qui peut être un morceau de texte, une partie d'une DTD, ou même une référence à un fichier textuel ou binaire extérieur. Il y a 2 sortes d'entités :

   - Les entités *internes* sont utilisées pour créer des raccourcis vers des parties de texte fréquemment utilisées. Le texte de remplacement est défini dans la déclaration de l'entité.

     ```
     <!ENTITY ATI "ArborText, Inc.">
     ```

     Ainsi, partout dans le document, la commande `&ATI;` insèrera la chaîne *"ArborText, Inc."*. Les spécifications XML ont déjà prédéfinies cinq entités internes (`lt, gt, amp, apos, quot`).
   - Les entités *externes* sont utilisées pour faire référence à des fichiers extérieurs. Ce fichier peut être textuel et son contenu sera inséré à l'emplacement de la commande `&<nom_entité>;` ou binaire et alors le fichier sera passé à une application.

     ```
     <!ENTITY boilerplate SYSTEM "/standard/legalnotice.xml">
     <!ENTITY ATIlogo     SYSTEM "/standard/logo.gif" NDATA GIF87A>
     ```

4. La déclaration de notations désigne par un nom le format d'une entité non-analysable, le format d'un élément muni d'un attribut notation ou l'application cible d'une instruction de traitement. Ainsi, une notation identifie par exemple des types spécifiques de fichiers binaires extérieurs. Une déclaration typique est alors :

   ```
   <!NOTATION GIF87A SYSTEM "GIF">
   ```

2.2.3 RDF

L'avenir du Web tel que le conçoit le W3C passe par une structuration des données qu'il contient. XML fournit en cela un excellent support puisqu'il permet d'obtenir un arbre représentant la structure du document et par conséquent sa sémantique d'une certaine façon. RDF (*Ressource Description Framework*) [LS99] est aussi une recommandation du W3C depuis février 1999. Il fournit un moyen pour annoter un document. C'est une application de XML dans le sens où la syntaxe de RDF est définie en XML.

RDF est donc un langage permettant de décrire les données d'un document, c'est-à-dire d'attacher des propriétés aux informations transportées. Le principe de base est de qualifier les données qu'il manipule en les déclarant en tant qu'entité. Ensuite, il est possible d'effectuer diverses opérations sur ces entités, principalement de leur attacher des propriétés et de les lier entre elles par diverses relations [Heb99]. Le résultat obtenu peut être représenté par des triplet sous la forme :

$$ressource \rightarrow propriété \rightarrow valeur$$

L'exemple suivant indique que l'auteur du document pointé par l'URI est `Raphaël` :

```
<rdf:description about="http://escrire.inrialpes.fr/" author="Raphaël"/>
```

- Une *ressource* est une entité manipulée dans une annotation RDF et accessible par une URI. Il existe un moyen pour spécialiser et classifier en différentes catégories ces ressources.

- Une *propriété*, comme son nom l'indique, permet d'attacher des informations à une ressource. C'est donc une relation binaire entre des ressources et/ou des valeurs atomiques. Tout comme pour les ressources il est possible de spécialiser les propriétés.

- Une *valeur* peut être simplement une chaîne de caractères, ou alors une ressource.

Cette technique des triplets permet aussi de créer des collections, c'est-à-dire des groupes de ressources auxquels s'appliquent diverses propriétés. La réification est enfin possible en transformant un triplet en ressource. Ce procédé permet donc d'attacher une propriété à un triplet.

Pour que cette information soit utilisable par une machine, il faut auparavant la définir. Ceci est le rôle de RDF-Schemas [BG00], bientôt au stade de recommandation, qui permet de déclarer des classes de ressources, et d'indiquer la signature des propriétés, c'est-à-dire de contraindre les domaine et co-domaine des relations.

Nous venons donc de voir différentes manières permettant de représenter le contenu de documents. XML, est en réalité un méta-langage et semble donc bien adapté puisqu'il permet de créer d'autres langages pour représenter la connaissance, à la différence de RDF qui contraint d'emblée le domaine des possibilités.

2.3 Langages de requêtes

D'une façon générale, la connaissance contenue au sein d'un document, une fois formalisée, peut être vue comme la conjonction de couples `(attribut,valeur)` dans un certain domaine, et d'autres éléments représentant le contenu. Par conséquent, une requête sur un corpus de documents sera la conjonction de contraintes portant sur les attributs du document, et d'autres sur son contenu. [Mai98] et [ESC00] ont isolé certaines spécifications concernant les requêtes :

- la *sélection* : choisir un document ou un élément de document en se basant sur son contenu, sa structure ou ses attributs ;

- l'*extraction* : extraire une sous-partie d'un document ;

- la *réduction* : restreindre un élément retenu en enlevant certains de ses sous-éléments ;

- la *restructuration* : construire un nouvel ensemble d'éléments satisfaisant la requête ;

- la *combinaison* : fusionner deux ou plusieurs éléments en un seul.

Nous allons décrire deux langages de requêtes, tous deux déclaratifs. Le premier, OQL (section 2.3.1), est un dérivé du langage SQL appliqué aux bases de données à objets. Il profite donc d'une longue expérience d'interrogation de données contrairement au second qui n'est encore qu'au stade de construction. XML-QL (section 2.3.2), bien que conduit par des gens issus des bases de données, est en effet une proposition de langage pour pouvoir interroger des données XML menée par le W3C.

2.3.1 OQL : le langage de requête standard de l'ODMG

L'ODMG (*Object Data Managment Group*) [ODM] est une référence dans le monde des bases de données à objets. Ce groupe a développé entre autre un langage d'interrogation de type déclaratif. OQL (*Object Query Language*) est basé sur SQL-92. Bien que conçu pour interroger des bases de données à objets, il semble adapté à notre problème d'autant plus qu'il profite d'une longue exprérience dans le domaine des requêtes. Il repose sur les principes suivants :

- il est cohérent avec le modèle objet de l'ODMG ;
- il ressemble fort à SQL-92 avec des extensions orientées objet ;
- il fournit des procédures de haut niveau pour traiter des ensembles d'objets, mais peut traiter également des structures, des listes et des tableaux avec autant d'efficacité ;
- c'est un langage fonctionnel : on peut composer opérateurs et opérandes librement tant qu'on obtient une expression valide, c'est-à-dire respectant le typage ;
- il utilise un accès déclaratif aux objets ce qui permet d'optimiser facilement les requêtes ;
- il possède une sémantique formelle facile à définir.

OQL est donc un langage pûrement fonctionnel qui accepte (théoriquement !) la syntaxe de SQL [ODM].

On peut accéder à un objet stocké dans une base de données grâce au nom de l'objet. De manière plus générale, on peut « naviguer » à travers un objet pour en extraire les propriétés que l'on veut. Pour cela, on utilise la notation "." (ou "→"). Soit par exemple une personne p, et on veut connaître le nom de la ville où habite son épouse. On écrira :

```
p.épouse.adresse.ville.nom
```

Voici un petit exemple de requete compte tenu de la taxonomie suivante (Figure 2.3) :

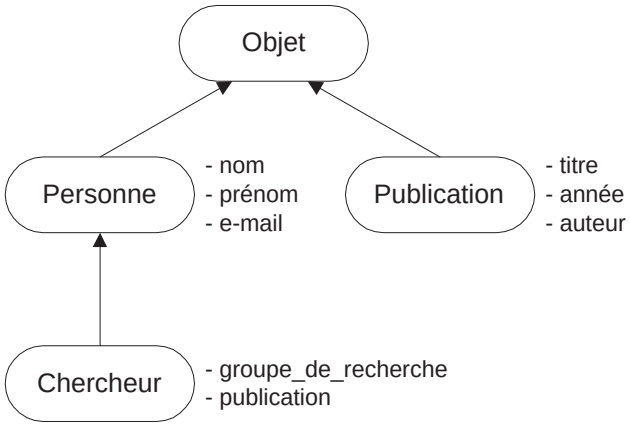

FIG. 2.3 – *Exemple d'une taxonomie de classes avec leurs attributs*

On désire connaître l'ensemble des publications de l'action de recherche EXMO avec pour chaque publication, le nom de l'auteur et son adresse e-mail?

```
SELECT c.publication.titre, c.nom, c.e-mail
FROM Chercheur c
WHERE c.groupe_de_recherche = "EXMO"
```

Cette requête va retourner une liste de documents (les publications) classée pertinemment (ce qui est encore un autre problème), mais aussi une valeur pour les autres variables de la requête.

2.3.2 XML-QL : un groupe de travail du W3C

Avec le développement de XML, le W3C a formé un groupe de travail pour développer des techniques permettant de formuler des requêtes sur des données XML. Une proposition récente est *XQuery* (ou *XML Query Language*) [Mai98] qui consiste à formuler des requêtes en XML. Les opérations élémentaires (*sélection*, *extraction*, *réduction*, *restructuration*, et *combinaison*) sont possibles en une seule requête et [Mai98] illustre avec un exemple ces différentes opérations sur un ensemble de données au format XML.

Le document de travail actuel du W3C [FMR00] a pour but de fournir un ensemble d'opérateurs permettant d'interroger des documents XML, ainsi que le langage basé sur ces opérateurs. Même si la syntaxe de ce langage n'est pas encore fixée, le groupe de travail a déjà posé un certain nombre d'exigences concernant ce futur langage. Ainsi, les requêtes pourront porter sur un seul document, une partie d'un document ou sur une collection de documents répondant ainsi aux critiques formulées par [MMA98]. Elles seront capables d'engendrer de nouveaux documents XML avec éventuellement leurs schémas (DTD) et mettront à profit l'utilisation de *XLinks* et de *XPointers*. Enfin, l'ordre des éléments qui a été fourni dans les documents XML, et donc l'association de ces éléments, peut être préservé par les requêtes.

Cependant, on peut encore reprocher quelques défauts à XML-QL. En effet, le document de travail indique qu'une requête pourra être formulée sur n'importe quel document XML, qu'il soit ou non valide vis à vis d'un schéma. Mais comme le signale [MMA98], l'absence de schéma pour des documents XML peut se révéler une grande carence au moment de formuler des requêtes. Aussi, on peut considérer que les documents XML doivent absolument être associés à un schéma, et que les requêtes doivent exploiter le plus possible cette structure.

De plus, et c'est sans doute la carence la plus importante pour l'objectif que nous poursuivons, XML-QL ne sert qu'à interroger la structure d'un document. Cela constitue déjà une avancée notable par rapport à une recherche traditionnelle « *en texte intégral* », mais on est encore loin d'atteindre la sémantique d'un document. Prenons un exemple simple :

```
<pere name="a">
      <fils name="b" />
</pere>
<pere name="b">
      <fils name="c" />
</pere>
```

Cet exemple indique que `a` est le `pere` de `b` qui est lui-même le `pere` de `c`. Imaginons maintenant qu'une requête pose la question : est-ce que `a` est le `grand-pere` de `c` ? Un langage comme XML-QL ne peut pas répondre à ce genre de requête car il ne connaît même pas la notion de `grand-pere`.

Ce dernier exemple nous montre finalement que ce n'est pas la façon dont les requêtes sont formulées qui est importante, mais plutôt comment elles vont être interprétées en exploitant la connaissance représentée. Le langage de requêtes choisi devra sans doute rester proche de standards comme les travaux actuels du W3C.

2.4 Systèmes intégrés

Nous avons déjà constaté l'étendue et la richesse des informations présentes sur le Web. Mais, en règle générale, cet ensemble d'informations n'est pas (ou très peu) structuré. Cette liberté laissée aux concepteurs de documents est d'ailleurs dans une large mesure responsable du succès du Web. Mais cette liberté introduit aussi de sérieux défauts lorsque l'on souhaite rechercher de l'information.

Pour parer à ces limitations, différents systèmes ont essayé de décrire la sémantique des informations contenues dans les documents afin d'améliorer les réponses aux requêtes formulées. La syntaxe de ces réponses est parfaitement définie et permet donc à des agents d'effectuer des traitements automatiques. De plus, ils se proposent de développer un véritable moteur d'inférence capable de compléter leur base de faits. Parmi ces systèmes, on peut citer OIL (*Ontology Interchange Language*) [HFB+00], XOL (*An XML-Based Ontology Exchange Language*) [KCT99], OML (*Ontology Markup Language*) [Ken]... Nous allons décrire deux de ces langages, souvent cités car relativement aboutis : SHOE (section 2.4.1) et ONTOBROKER (section 2.4.2).

2.4.1 SHOE (Simple HTML Ontology Extensions)

SHOE ([LSRH97], [HHL99]) est un projet développé à l'université de Maryland par Jeff Heflin, Sean Luke, Lee Spector, David Roger et James Hendler. Ce projet a vu le jour en 1996. SHOE se veut un langage de représentation de connaissances qui permet de construire des ontologies et de les utiliser. Il est donc basiquement constitué d'*ontologies* et d'*instances* :

- une ontologie va définir les règles indiquant quels types d'assertions et d'inférences on peut faire ;
- une instance va déclarer des assertions basées sur ces règles.

C'est un langage compatible avec HTML et XML qui permet donc de définir des ontologies ou alors d'étendre des versions déjà existantes. Il permet aussi de déclarer des entités et de les classifier. Enfin des relations n-aires entre les entités peuvent être définies et des règles d'inférence créées. Les ontologies et les instances sont séparées et possèdent un identificateur unique. Nous allons maintenant décrire la sytaxe de SHOE ainsi que les outils dont il dispose.

Définition des ontologies

Une ontologie doit être décrite entre les tags `<ONTOLOGY ID=`*`id`* `VERSION=`*`version`*`>` et `</ONTOLOGY>`. Le couple *<id,version>* définit l'identificateur unique de l'ontologie. Avec SHOE, une ontologie peut dériver d'une ou plusieurs autres super-ontologies (dans un schéma d'héritage multiple) ou mettre à jour des versions plus anciennes de la même ontologie. SHOE définit les types basiques *strings, numbers, dates* et *boolean* (bientôt *URI*). Une ontologie peut définir des catégories (ou classes) (`<DEF-CATEGORY>`) et les classifier dans la taxonomie, toujours sous le type le plus général `SHOEEntity`. Elle peut définir des relations n-aires (`<DEF-RELATION>`) entre des instances de catégories, et d'autres instances ou des types basiques. Enfin, une ontologie peut contenir des règles d'inférences (`<DEF-INFERENCE>`) de type clauses de Horn.

L'exemple suivant résume les différentes possibilités de SHOE lors de la construction d'une ontologie :

```
<HTML>
<HEAD>
      <TITLE>University Ontology</TITLE>
```

```
<!-- On indique que ce document est conforme avec la spécification 1.0 de SHOE -->
        <META HTTP-EQUIV="SHOE" CONTENT="VERSION=1.0" />
  </HEAD>
  <BODY>
<!-- On déclare le nom de l'ontologie, sa version, et les versions compatibles -->
        <ONTOLOGY ID="university-ontology" VERSION="1.0" BACKWARD-COMPATIBLE-WITH="0.5 0.7">
<!-- On déclare qu'on utilise une autre ontologie, dont les élément seront préfixés par "base" -->
        <USE-ONTOLOGY ID="base-ontology" VERSION="1.0" PREFIX="base"
                URL="http://www.cs.umd.edu/projects/plus/SHOE/base.html" />
<!-- On définit des catégories et leur organisation hiérarchique -->
              <DEF-CATEGORY NAME="Person" ISA="base.SHOEEntity" />
              <DEF-CATEGORY NAME="Organization" ISA="base.SHOEEntity" />
              <DEF-CATEGORY NAME="Worker" ISA="Person" />
              <DEF-CATEGORY NAME="Advisor" ISA="Worker" />
              <DEF-CATEGORY NAME="Student" ISA="Person" />
              <DEF-CATEGORY NAME="GraduateStudent" ISA="Student Worker" />
<!-- On définit des relations, ici binaires, mais qui peuvent être n-aires -->
              <DEF-RELATION NAME="advices">
                    <DEF-ARG POS=1 TYPE="Advisor" />
                    <DEF-ARG POS=2 TYPE="GraduateStudent" /></DEF-RELATION>
              <DEF-RELATION NAME="age">
                    <DEF-ARG POS=1 TYPE="Person" />
                    <DEF-ARG POS=2 TYPE="base.NUMBER" /></DEF-RELATION>
              <DEF-RELATION NAME="suborganization">
                    <DEF-ARG POS=1 TYPE="Organization" />
                    <DEF-ARG POS=2 TYPE="Organization" /></DEF-RELATION>
              <DEF-RELATION NAME="works-for">
                    <DEF-ARG POS=1 TYPE="Person" />
                    <DEF-ARG POS=2 TYPE="Organization" /></DEF-RELATION>
<!-- On définit enfin des règles d'inférence -->
              <DEF-INFERENCE>
                    <INF-IF>
                          <RELATION NAME="works-for">
                                <ARG POS=1 VALUE="x" VAR />
                                <ARG POS=2 VALUE="y" VAR /></RELATION>
                          <RELATION NAME="suborganization">
                                <ARG POS=1 VALUE="y" VAR />
                                <ARG POS=2 VALUE="z" VAR /></RELATION></INF-IF>
                    <INF-THEN>
                          <RELATION NAME="works-for">
                                <ARG POS=1 VALUE="x" VAR />
                                <ARG POS=2 VALUE="z" VAR /></RELATION></INF-THEN>
              </DEF-INFERENCE>
        </ONTOLOGY>
  </BODY>
  </HTML>
```

Définition des instances

Une instance est d'abord un objet au sens des objets dans une base de données. C'est aussi un ensemble de faits qui alimentent une base de connaissance. Ce sont ces faits qui apporteront les réponses à des requêtes structurées. Une instance apparaît entre les tags `<INSTANCE Key=`*clé*`>` et `</INSTANCE>`, où *clé* est l'identificateur unique de l'instance (en général, l'URI de la page Web). Une instance fait référence à une ou plusieurs ontologies, et va donc contenir des assertions portant sur les catégories et les relations définies dans celles-ci.

L'exemple suivant est une page Web personnelle, instance de l'ontologie définie précédement :

```
<HTML>
<HEAD>
      <TITLE>John's Web Page</TITLE>
<!-- On indique que ce document est conforme avec la spécification 1.0 de SHOE -->
      <META HTTP-EQUIV="SHOE" CONTENT="VERSION=1.0" />
</HEAD>
<BODY>
      <P>This is my home page. I've got some SHOE data on it about me and my advisor.</P>
<!-- On déclare la clé de l'instance -->
      <INSTANCE KEY="http://univ.edu/john">
<!-- On déclare qu'on utilise une ontologie en donnant sa location ; les élément sont préfixés par "uni" -->
            <USE-ONTOLOGY ID="university-ontology" VERSION="1.0" PREFIX="uni"
                  URL="http://univ.edu/ontology" />
<!-- On asserte des catégories -->
            <CATEGORY NAME="uni.GraduateStudent" />
            <CATEGORY NAME="uni.Advisor" FOR="http://univ.edu/mike" />
<!-- On asserte des relations -->
            <RELATION NAME="uni.advices">
                  <ARG POS=1 VALUE="http://univ.edu/mike" />
                  <ARG POS=2 VALUE=me /></RELATION>
            <RELATION NAME="uni.age">
                  <ARG POS=1 VALUE=me />
                  <ARG POS=2 VALUE="32" /></RELATION>
      </INSTANCE>
</BODY>
</HTML>
```

Outils associés à SHOE

Autour du langage SHOE, l'équipe a developpé un agent EXPOSÉ qui parcourt les pages Web à la recherche d'informations sémantiques. Dès qu'il remarque une page annotée, il charge la ou les ontologies utiles et récolte les données pour compléter la base de connaissance.

L'équipe a aussi développé un outil pour annoter graphiquement des pages Web. Cette tâche est sans doute la plus fastidieuse pour un utilisateur voulant fournir de la connaissance. De plus, elle nécessite une connaissance du formalisme SHOE. Grâce à cet outil, l'annotation de document devient semi-automatique. Enfin, un dernier outil permet à des utilisateurs de formuler des requêtes via une interface graphique.

Bilan

Pour démontrer les capacités de SHOE qui ont été présentées, deux applications ont été contruites.

La première application concerne le domaine des départements d'informatique dans les universités. Des catégories (`Student, Faculty, Course, Department, Publication, Research...`) et des relations (`publicationAuthor, emailAddress, advisor, member...`) ont été définies et des pages annotées. L'agent EXPOSÉ a été utilisé pour récolter les informations et former la base de connaissance. Le résultat de cette expérience est assez concluant bien que la base obtenue soit très petite. La tâche la plus difficile finalement réside dans le choix de l'information à annoter. En effet, il semble présomptueux de vouloir formaliser toute la connaissance contenue dans une page Web, alors comment identifier les concepts pertinents qui vont être annotés ?

Une seconde application a permis de valider SHOE et concerne l'épidémie de la vache folle. Le but est de construire un site Web contenant des informations sur cette maladie et son lien

apparent avec la maladie de Creutzfeld-Jakob. La connaissance doit être accessible à un large public (chercheurs ou grand public) et donc, doit répondre à des questions dont la terminologie et la complexité diffèrent beaucoup. Cette seconde application a mis en relief d'autres problèmes que la première expérience n'avait pas montré. Tout d'abord, la difficulté de construire une ontologie efficace compte tenu de la nature même du domaine et de sa frontière mal définie. Le processus d'annotation des pages est lui aussi beacoup plus compliqué à cause des données. Enfin, il reste toujours à trouver le bon compromis entre le niveau de détail de la connaissance à donner et le temps que cela coûte pour annoter les pages.

2.4.2 ONTOBROKER

ONTOBROKER ([FDES98], [FES98], [FAD+98], [DEFS98]) est un projet développé à l'université de Karlsruhe par Rudi Studer, Stefan Decker, Michael Erdmann et Dieter Fensel. Ce projet a vu le jour en 1997. ONTOBROKER fournit différents langages pour décrire des ontologies, formuler des requêtes, ou encore annoter des documents. Nous allons donc décrire l'architecture de ce système et chacun des formalismes de représentation.

Architecture générale

L'architecture générale d'ONTOBROKER est décrite dans la Figure 2.4. Elle consiste en trois éléments principaux (une interface permettant d'effectuer des requêtes, un moteur d'inférence, et un « *webcrawler* »). Chacun de ces éléments est accompagné d'un langage de formalisation. Ainsi, ONTOBROKER fournit un langage pour représenter des ontologies. Un sous-ensemble de ce langage permet de formuler des requêtes sur les ontologies. Enfin, un dernier langage permet d'annoter les pages Web, c'est-à-dire de les enrichir avec des méta-informations décrivant la sémantique des documents compte tenu d'une certaine ontologie.

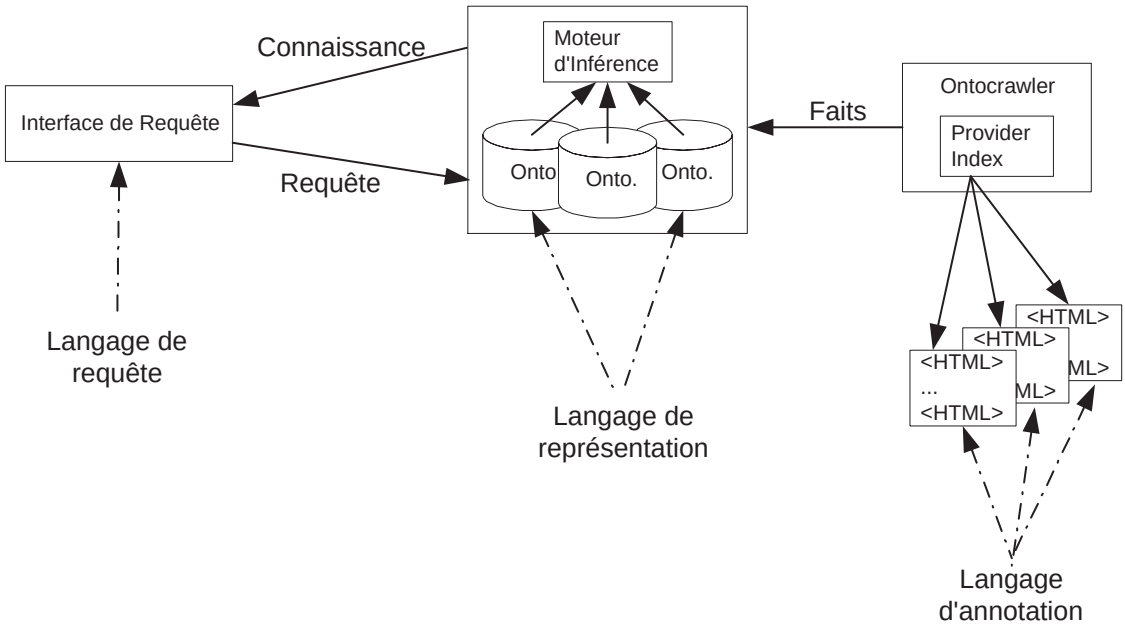

FIG. 2.4 – *Architecture générale d'*ONTOBROKER

L'utilisateur communique avec le moteur d'inférence via une interface permettant de formuler des requêtes. Le moteur utilise alors deux sources d'information pour dériver la réponse. Il utilise tout d'abord le schéma défini dans une ontologie, puis ensuite une base de faits collectés par le « webcrawler ». En, effet, un fournisseur d'informations (c'est-à-dire une personne qui a annoté des documents avec des méta-données) doit au préalable s'enregistrer en fournissant une liste de toutes les adresses URL contenant des pages annotées. Le « webcrawler », appelé « *ontocrawler* », parcourt ensuite ces pages et extrait les méta-informations pour constituer une base de faits utilisée par le moteur d'inférence.

Un langage pour exprimer des ontologies

Une ontologie va être décrite par une hiérarchie de classe (*taxonomie*) et un ensemble de règles (utilisées par le moteur d'inférence). ONTOBROKER utilise *Frame-Logic* pour modéliser des ontologies. Ce langage est basé sur le calcul des prédicats restreint aux clauses de Horn (type *Prolog*). Il permet entre autre de représenter et de hiérarchiser des concepts (ou classes), des instances ou des attributs avec leur valeur.

Un certain nombre de primitives élémentaires sont fournies :

- sous-classe : C1 : : C2, signifie que *C1* est une sous-classe de *C2* ;
- instance : O : C, signifie que *O* est instance de la classe *C* ;
- déclaration d'attribut : C1[A ==>>C2], signifie que les instances de *C1* ont un attribut *A* dont la valeur doit être une instance de *C2* ;
- valeur d'attribut : O[A —>>V], signifie que l'objet *O* a un attribut *A* de valeur *V* ;
- partie de : O1 < : O2, signifie que *O1* est une partie de *O2*.

De plus, des expressions plus complexes peuvent être construites à partir de ces expressions élémentaires et des connecteurs logiques habituels ($\rightarrow$, $\leftarrow$, $\leftrightarrow$, *AND, OR, NOT*). Ainsi,

- une règle à la forme :

  $$\text{tête} \leftarrow \text{corps}$$

  où : *tête* est une conjonction *(AND)* d'expressions élémentaires, et *corps* une expression complexe où l'on a éventuellement introduit des variables via les quantificateurs *FORALL* et *EXISTS* ;

- une règle double à la forme :

  $$\text{tête} \leftrightarrow \text{corps} \ (\simeq \text{tête} \leftarrow \text{corps AND corps} \leftarrow \text{tête})$$

  où : *tête* et *corps* sont des conjonctions d'expressions élémentaires.

Le Tableau 2.2 nous montre un exemple d'ontologie où l'on a défini tour à tour des classes (avec leurs attributs), les relations entre ces classes, et un ensemble de règles. La déclaration de la hiérarchie de classes est une suite d'expressions élémentaires décrivant les liens entre les classes. La définition des attributs indique les attributs et leurs types, des différents classes. La première règle assure la symétrie de la règle de coopération. La seconde spécifie que si une personne est connue pour avoir une publication, alors cette publication a un auteur qui doit être cette personne !

Classes	Attributs	Règles
$Object[\,].$ $Person :: Object.$ $Employee :: Person.$ $AcademicStaff :: Employee.$ $Researcher :: AcademicStaff.$ $Publication :: Object.$	$Person[$ $firstName \Rightarrow\!\!> STRING;$ $lastName \Rightarrow\!\!> STRING;$ $eMail \Rightarrow\!\!> STRING;$ ... $publication \Rightarrow\!\!> Publication].$ $Employee[$ $affiliation \Rightarrow\!\!> Organization;$ $worksAtProject \Rightarrow\!\!> Project;$ $headOfGroup \Rightarrow\!\!> ResearchGroup].$ $AcademicStaff[$ $supervises \Rightarrow\!\!> PhDStudent].$ $Researcher[$ $researchInterest \Rightarrow\!\!> ResearchTopic;$ $memberOf \Rightarrow\!\!> ResearchGroup;$ $cooperatesWith \Rightarrow\!\!> Researcher].$ $Publication[$ $author \Rightarrow\!\!> Person;$ $title \Rightarrow\!\!> STRING;$ $year \Rightarrow\!\!> STRING;$ $abstract \Rightarrow\!\!> STRING].$	$FORALL\ \ Person1, Person2$ $Person1 : Researcher$ $[cooperatesWith \rightarrow\!\!> Person2]$ $\leftarrow$ $Person2 : Researcher$ $[cooperatesWith \rightarrow\!\!> Person1].$ $FORALL\ \ Person1,$ $Publication1$ $Publication1 : Publication$ $[author \rightarrow\!\!> Person1]$ $\leftrightarrow$ $Person1 : Person$ $[publication \rightarrow\!\!>$ $Publication1].$

TAB. 2.2 – *Définition d'une ontologie dans* ONTOBROKER

Un langage pour exprimer des requêtes

Nous venons de voir comment définir des classes (concepts), des instances de classes (objets), des attributs et leurs valeurs. Un sous-ensemble de ce formalisme permet de formuler des requêtes. Globalement, une requête à la forme :

$$FORALL\ \ O \leftarrow O : C[A \rightarrow\!\!> V]$$

Cela signifie que O est un objet, instance de la classe C, dont un attribut A à la valeur V. Ainsi la requête suivante nous donne tous les objets instance de la classe chercheur :

$$FORALL\ \ R \leftarrow R : Researcher$$

On peut alors préciser la requête pour obtenir le nom des chercheurs dont le prénom est *Richard*, avec leur adresse e-mail :

$$FORALL\ \ Obj, LN, EM \leftarrow Obj : Researcher[firstName \rightarrow\!\!> Richard; lastName \rightarrow\!\!> LN; email \rightarrow\!\!> EM]$$

ONTOBROKER est aussi capable de faire des inférences sur l'ontologie.

$$FORALL\ \ Obj, CP \leftarrow Obj : Researcher[lastName \rightarrow\!\!> "Motta"; cooperatesWith \rightarrow\!\!> CP]$$

Cette requête est intéressante car l'ontologie contient une règle spécifiant la symétrie de la relation de coopération. Ainsi, même si le chercheur nommé *Motta* n'a pas indiqué de coopération, mais qu'un autre chercheur a lui spécifié qu'il collaborait avec *Motta*, le moteur d'inférence trouvera l'information. Enfin, ONTOBROKER peut formuler des requêtes sur l'ontologie elle-même. Ainsi la requête suivante retourne toutes les classes définies dans l'ontologie, qui sont sous-classes de la classe *Object* :

$$FORALL\ \ C \leftarrow C :: Object[\,]$$

Un langage pour annoter des documents

Actuellement, la plupart des documents sur le Web sont au format HTML. C'est pourquoi, une simple extension à ce langage est utilisée pour annoter les pages Web. Ainsi, toutes les méta

informations décrivant la sémantique du document sont contenues au sein même de la page. Le concepteur de document peut annoter ses pages graduellement et les navigateurs standards sont toujours capable de lire ces pages.

La déclaration d'une ontologie fournit déjà le schéma conceptuel, c'est-à-dire la définition des classes et de leurs attributs ainsi que les relations entre les classes. Par contre, les noms correspondants aux instances des classes et les valeurs des attributs doivent être indiqués à l'intérieur du document, grâce au langage qui permet de l'annoter. Pour cela, le concepteur dispose de trois primitives contenues à l'intérieur du tag spécifique <a ...>... et de l'attribut onto (le tag contient déjà les attributs *name* et *href*). Ainsi, le document reste lisible par les navigateurs standards (*Netscape, MSExplorer*). De plus, des raccourcis sont offerts pour rendre l'annotation de documents plus pratique.

– Un objet est identifié par une URI. Il peut être défini comme instance d'une classe. Par exemple, pour exprimer que Richard Benjamins est un chercheur, on écrira :

```
<a onto="‘http://www.iiia.csic.es/~richard‘:Researcher"></a> ou
<a onto="page:Researcher"></a>
```

– La valeur d'un attribut d'un objet peut être fixé. Par exemple, pour donner l'adresse e-mail du chercheur Richard Benjamins, on écrira :

```
<a onto="‘http://www.iiia.csic.es/~richard‘[email=‘mailto:richard@iiia.csic.es‘]"></a>
```

On peut aussi écrire :

```
<a onto="page[firstName=body]">Richard</a>
```

qui indique que Richard est bien la valeur de l'attribut de l'objet identifié par *page*.

– Enfin, une relation entre plusieurs objets peut être établie.

```
<a onto="appointment(
                page,
                'http://aifb.uni-karlsruhe.de/WBE/~dfe',
                'October, 17th 1997',
                body)">
      (KA)2-Initiative: The Inaugural Meeting</a>
```

Cet exemple défini une relation *appointment* avec quatre attributs : deux personnes identifiées par la page actuelle et une autre URI, une certaine date et un nom dont la valeur se trouve dans le corps du tag.

Les outils dont dispose ONTOBROKER

Pour utiliser ces différents langages, ONTOBROKER est composé de trois outils : une interface utilisateur, un moteur d'inférence et un « webcrawler » qui se charge de collecter les informations sémantiques contenues dans les pages Web annotées et de les transformer en faits.

Pour formuler des requêtes, l'utilisateur dispose de deux interfaces :

– Une interface en mode texte destinée aux utilisateurs experts, permet de saisir directement des requêtes, mais elle suppose qu'ils connaissent d'une part la syntaxe du langage permettant d'exprimer des requêtes et d'autre part qu'ils connaissent l'ontologie !

– Une interface graphique destinée aux novices fournit elle, un graphe hyperbolique contenant toutes les classes et les liens entre elles. Lorsque l'utilisateur clique sur une classe, celle-ci apparait dans une seconde fenêtre et il peut alors entrer des valeurs pour les atributs possibles. Une requête est alors automatiquement générée dans le formalisme décrit ci-dessus.

Le moteur d'inférence est chargé de dériver les réponses aux requêtes formulées. Il utilise pour cela les faits (contenus dans les pages Web) et la définition de l'ontologie. Une première phase consiste donc à traduire en prédicats, l'ontologie déclarée en *F-Logic* (Tableau 2.3). Le moteur d'inférence combine alors les faits et les règles obtenus et résoud le problème donné en logique des prédicats par substitution et unification. Une dernière transformation est effectuée pour retourner la réponse à la requête en *F-Logic*.

Frame-Logic	Sens	Logique des prédicats
$C1 :: C2$	la classe *C1* est sous-classe de la classe *C2*	$sub(C1, C2)$
$O : C$	O est une instance de la classe C	$isa(O, C)$
$C1[A \Rightarrow\!\!> C2]$	pour toute instance de la classe *C1*, un attribut A est défini et doit avoir comme valeur une instance de la classe *C2*	$att\_type(C1, A, C2)$
$O[A \rightarrow\!\!> V]$	l'instance O a un attribut A de valeur V	$att\_val(O, A, V)$
$O1 <: O2$	*O1* est une partie de *O2*	$part\_of(O1, O2)$

TAB. 2.3 – *Transformation de F-Logic en logique des prédicats*

Bilan

Deux inconvénients majeurs apparaissent avec cette approche. Tout d'abord, une ontologie doit être définie. Cela suppose qu'un groupe de personnes doit se mettre d'accord sur les informations à représenter. Cette étape est donc complètement dépendante du contexte. Cependant, il paraît évident qu'un tel modèle, une fois construit, améliore considérablement la présentation des documents Web, car ceux-ci seront sans doute mieux structurés. De plus, il pourra être ré-utilisé par des compagnies comme une référence. Ensuite les documents doivent être annotés. Le choix d'inclure ces méta-informations au sein même des pages HTML évite au maximum la redondance d'information. En effet, le concepteur n'est pas obligé d'écrire une première fois son document et une seconde fois son document formalisé. Cependant, une interface graphique pourrait être utilisée pour annoter automatiquement les documents, de la même façon que pour formuler des requêtes. ONTOBROKER prévoit d'ajouter cette fonctionnalité.

ONTOBROKER avait pour but de permettre un meilleur accès à l'information dans une gigantesque base de connaisances que serait le Web. Actuellement, deux ontologies assez importantes sont disponibles. L'une est utilisée par la communauté de l'acquisition de connaissances $(KA)^2$ (*Knowledge Annotation Initiative of the Knowledge Acquisition Community*). Elle décrit des organisations (entreprise, université, institut de recherche), des personnes (employé, étudiant, administration), des projets et des évènements. L'autre provient du *CIA World Fact Book* et concerne l'organisation géographique, démographique, économique et politique de tous les pays du monde. En trois ans de fonctionnement, l'équipe d'ONTOBROKER a constaté un certain nombre de défauts à leur approche :

– L'URI a été choisie pour identifier de manière unique les objets. Cependant, plusieurs URI peuvent pointer sur la même page. Quelle URI faut-il alors choisir ? De plus, dès lors

qu'une URI fait référence à une page, fait-elle référence à la page ou à ce dont elle contient (université, chercheur...)?

- ONTOBROKER a été aussi confronté à un problème de performances. En effet, lorsque le moteur d'inférence traduit les ontologies en logique, seulement quelques prédicats sont utilisés (Voir Tableau 2.3) pour exprimer la hiérarchie des classes et leurs attributs. Il en résulte un nombre restreint de prédicats, impliqués dans un nombre de règles toujours plus important au fur et à mesure que l'ontologie grossit et donc un problème de performances lorsqu'il s'agit d'unifier les termes.
- L'unification elle-même pose des problèmes. Par définition, elle vérifie que deux termes sont littéralement égaux. Ainsi, une simple faute d'ortographe dans une requête, ou pire encore dans l'annotation d'une page, retournera des mauvaises réponses.

Ces problèmes commencent à être résolu dans une évolution d'ONTOBROKER appelée ON2BROKER.

2.5 Conclusion

Le but de notre étude est de représenter la connaissance contenue dans des documents afin d'indexer ensuite ces documents par leur sens. Nous avons d'ailleurs présenté deux expériences qui essaient de décrire la sémantique des informations contenues dans un document et qui ont donné lieu à un langage. Le premier, SHOE, se présente comme une simple extension du langage HTML et notamment de l'élément `<META>`. Le second, ONTOBROKER, s'inspire plutôt de l'élément `<SPAN>` disponible depuis la version 4 de HTML. A la différence de SHOE, ONTOBROKER a décidé d'inclure ses annotations directement dans le code HTML de la page Web, évitant ainsi une redondance d'information. Ces deux systèmes offrent la possibilité de construire une ontologie et d'annoter des documents en y faisant référence. Ils proposent aussi un langage de requêtes avec une syntaxe et une sémantique spécifiées permettant ainsi à un agent d'utiliser les résultats. Enfin, un moteur d'inférence est capable de compléter la base des faits. Cependant, ces systèmes ont été conçu avec un but d'application précis. Aussi, lorsque les concepteurs de ces langages ont voulu les appliquer à un autre contexte, ils se sont heurtés à des problèmes de type quel niveau de détail donner au moment de la formalisation de la connaissance ou comment construire l'ontologie. Nous nous positionnons dans ce contexte comme un concurent de ces systèmes.

Nous avons étudié différents formalismes de représentation de connaissances. Ces derniers sont en effet destinés au stockage et à la manipulation des connaissances dans la machine. Parmi eux, les formalismes à objets apparaissent comme un moyen particulièrement adapté à la représentation des connaissances complexes sur un domaine en cours d'étude. Les unités de représentation de base sont les objets qui dénotent des entités individuelles du domaine. Ils sont décrits en termes d'attributs qui modélisent leurs propriétés et leurs relations à d'autres objets. Les objets sont organisés en classes qui expriment les catégories du domaine. Au sein d'une classe, les objets partagent des caractéristiques, exprimées par les attributs de la classe. Les classes sont organisées en une structure hiérarchique, la taxonomie, selon une relation de généralité appelée spécialisation. Nous disposons du système TROEPS, un représentant de la RCO. Il nous faut donc maintenant identifier quels sont les éléments d'un document à représenter.

Le W3C a spécifié dernièrement un langage universel, ou plutôt un méta-langage permettant de créer son propre langage. Nous avons donc présenté rapidement XML et le principe d'une DTD. La similarité de XML avec les systèmes à objets a été remarquée [Euz00] et ce langage devient donc un passage obligé pour la transcription d'un formalisme de représentation de connaissances par objets. Cependant, XML souffre d'une absence de sémantique. Le W3C a alors créé le langage

RDF basé sur XML et qui permet de décrire les données d'un document. Mais là encore, on n'atteint pas la sémantique des documents. En particulier, il est toujours impossible de faire des inférences. Si la cible que nous visons est donc clairement le langage HTML (ou plutôt XHTML), il nous faut résoudre le problème de l'intégration des annotations avec le document lui-même.

Un autre aspect de notre étude concerne la possibilité d'effectuer des requêtes sur les documents de notre corpus une fois que ceux-ci auront été indexé par leur sens. Nous avons donc présenté deux langages de requêtes. Le premier, OQL, a déjà fait ses preuves dans le domaine des bases de données à objets. Mais à la différence des bases de données où le schéma est très peu mis à contribution, l'ontologie risque d'être souvent manipulée. Le second, XML-QL, n'est encore qu'au stade d'un groupe de travail, mais il est d'ores et déjà intéressant de suivre les choix et les orientations de ce langage destiné à interroger des documents XML. Il nous reste donc à préciser quels types de requêtes un utilisateur est susceptible de poser.

Le chapitre suivant récapitule les questions que l'on doit se poser et tente d'apporter à chaque fois une réponse.

Chapitre 3

Quelles représentations pour quels contenus ?

Nous venons de présenter différents formalismes de représentation de connaissance. Parmi eux, la représentation de connaissances à objets apparaît particulièrement adapté lorsqu'il s'agit de représenter des connaissances complexes sur un domaine en cours d'étude. En effet, les unités de repésentation de base sont des objets et sont donc proches de la vision que l'on a d'un domaine. Mais indépendamment du formalisme utilisé, comment isoler les éléments de connaissance qui vont faire l'objet d'annotation ?

Nous allons commencer par présenter le corpus de travail utilisé tout le long de notre étude (section 3.1). Nous essaierons alors de voir comment on peut représenter la connaissance contenue dans un document (section 3.2). L'application résultante doit permettre de retrouver ces documents qui auront été indexés par leur sens. Nous verrons donc quels types de requêtes un utilisateur est susceptible de poser, et notamment quelles interprétations leurs donner (section 3.3). Nous étudierons enfin quels sont les moyens s'offrant à nous qui permettent d'intégrer la formalisation obtenue au document (section 3.4).

3.1 Présentation du corpus de travail

Le corpus de travail qui a été utilisé tout le long de notre étude concerne la drosophile et plus particulièrement les interactions géniques qui conduisent au processus de segmentation. La drosophile (ou mouche du vinaigre) est une petite mouche (environ 3 mm de longueur), que l'on rencontre sur les fruits en décomposition. L'espèce *Drosophila melanogaster* est un organisme modèle (parmi les mieux étudiés à l'heure actuelle) utilisé pour la recherche en biologie, en particulier dans le domaine de la génétique. Les grandes facilités d'élevage de la drosophile ont contribué à son immense succès. La drosophile possède en outre des chromosomes géants dans les cellules de ses glandes salivaires. Ce sont des chromosomes facilement observables au microscope, dont l'ADN est répliqué un très grand nombre de fois. Ces chromosomes sont rayés de bandes sombres et de bandes claires, ce qui permet un repérage très précis des gènes sur l'ADN. Un projet de cartographie et de séquençage de la totalité du génome de la drosophile est actuellement conduit par la communauté drosophiliste internationale. Depuis la fin du mois de mars dernier, cet objectif semble atteint, terminant plus rapidement que prévu le séquençage intégral du génome de la mouche.

L'ensemble des documents qui forment notre corpus provient de la base Medline [1] (*National*

1. http://www.ncbi.nlm.nih.gov/PubMed

Center for Biotechnology Information). Ce sont des résumés d'articles scientifiques obtenus à partir de requêtes sur la base de données. Une étape de notre étude consiste à annoter chacun de ces articles en intégrant une représentation formelle de leur contenu dans un langage de représentation de connaissances. Cette représentation du contenu va alors nous permettre d'effectuer des requêtes structurées et donc de rechercher et d'interroger des documents de façon plus efficace que ne le permet aujourd'hui la recherche plein texte par exemple. Cependant, si notre objectif est clairement de pouvoir rendre accessible à une machine la connaissance contenue dans un document, quels sont alors les moyens pour y parvenir ? En particulier, comment choisir les éléments pertinents à représenter ? Les sections suivantes tentent d'apporter quelques réponses aux questions que l'on doit se poser.

3.2 Comment représenter la connaissance contenue dans un document ?

Comment représenter la connaissance contenue dans un document ? Outre le choix du formalisme de représentation de connaissance à utiliser, on doit d'abord se demander à quel domaine appartiennent les documents composant notre corpus. On doit aussi savoir identifier les éléments pertinents qui vont être annotés. En effet, s'il semble tentant de vouloir représenter toute la connaissance contenue dans un document, est-ce que c'est bien raisonnable, et surtout, est-ce que c'est utile ? Sinon, comment juger de la pertinence d'un élément et décider si sa sémantique sera connue ou pas de la machine ?

Nous allons donc tout d'abord discuter de la nature des documents de notre corpus (section 3.2.1). Nous verrons alors quels constructeurs de représentation de connaissance seront utiles (section 3.2.2). Nous essaierons ensuite d'identifier quels éléments du document on va décrire en proposant plusieurs solutions possibles (section 3.2.3). Nous présenterons enfin les choix effectués en ce qui concerne la description de l'ontologie et donc quels éléments seront annotés dans les documents (section 3.2.4).

3.2.1 La nature des documents

Représenter la connaissance contenue dans un document est tout d'abord très lié à son type. En effet, prenons l'exemple d'un roman : toute l'histoire se réfère à un monde fictif qui est décrit dans l'œuvre. Dans le cadre d'une formalisation de la connaissance, on décrirait traditionnellement ce monde dans l'ontologie, mais il serait différent pour chaque roman. Ceci révèle donc l'extrême difficulté à vouloir annoter un ensemble de documents de fiction et même à l'interroger puisqu'une connaissance de ce monde fictif est nécessaire. En revanche, pour un document qui n'est pas une fiction comme par exemple un article scientifique, toutes les informations présentes dans le document se réfèrent à une partie d'un monde « universel » ou tout du moins partagé par les lecteurs. Le fait de décrire cette partie du monde permet donc d'annoter un ensemble de documents qui feront référence aux mêmes objets.

Dans notre étude, les documents de notre corpus ne seront clairement pas des œuvres de fiction. Ainsi, si un article signale l'influence d'un certain gène `X` sur un autre gène `Y`, il considère comme acquis l'existence même de ces deux gènes.

3.2.2 Quels constructeurs de représentation de connaissance faut-il utiliser ?

Nous proposons de représenter le contenu de documents grâce à un formalisme de représentation de connaissance. Mais quels constructeurs conviennent à cette tâche ? Si on se place dans

un contexte général de comparaison de formalismes de représentation de connaissances, qui est l'objectif de l'action ESCRIRE, il semble nécessaire de pouvoir disposer d'objets et de relations. Pour notre application, un formalisme de représentation de connaissances par objets a été choisi. Le langage doit donc être capable de décrire les objets d'un document, de valuer leurs différents attributs, et d'indiquer leur appartenance à des classes. Il doit aussi être capable de décrire des classes et de les organiser en une taxonomie. Enfin, les relations seront vues commes des objets.

Les documents qui vont faire l'objet d'annotations sont des articles scientifiques traitant des interactions géniques chez la drosophile. Ces documents contiennent un certain nombre d'assertions concernant la mise en évidence, par des expériences, d'interactions entre des gènes pendant le processus de segmentation de la mouche. Il est probable par conséquent que différents documents du corpus divergent sur quelques points, voire même se contredisent simplement parce que les expériences n'auront pas été réalisées dans les mêmes conditions et que ces conditions ne sont pas explicitement détaillées. On peut donc se demander si l'on doit pour chaque document représenter les faits tels qu'ils sont décrits, ou si l'on doit représenter le fait que l'auteur a déclaré dans tel article que certaines interactions ont été observées suite à ses expériences. Cette nuance s'avère en fait une réelle différence puisque dans le premier cas, on suppose que certains faits peuvent se contredire, alors que dans le second, c'est impossible (sauf si un auteur se contredit dans un même article !). Pour notre application, on se place dans un point de vue non épistémique dans la mesure où l'on représente les interactions telles qu'elles sont décrites dans les documents sans se soucier de ce que contiennent les autres documents du corpus.

3.2.3 Quels éléments du document va-t-on décrire ?

Après avoir identifié les constructeurs de représentation de connaissances utiles, il reste à déterminer quels sont les éléments qui vont justement faire l'objet d'annotation. Nous allons donc essayer de formaliser l'intégralité d'un document pour se rendre compte de la difficulté d'une telle démarche. Nous verrons ainsi que cette technique ne résoud pas tous les problèmes puisque la machine sera toujours incapable de répondre à un certains nombres de requêtes. Nous essaierons donc d'imaginer quelles alternatives se présentent à nous.

Une représentation exhaustive du document

L'objectif étant de rendre accessible à une machine le sens d'un document, la première idée qui vient à l'esprit est d'essayer de représenter toute la connaissance contenue dans le document. Une technique naturelle consiste alors à effectuer une analyse lexicographique du document, phrase par phrase, et d'essayer de représenter le contenu de celles-ci. Mais peut-on alors affirmer que toute la connaissance contenue dans le document a été représentée ? En particulier, qu'en est-il exactement de toutes les informations implicites qu'on peut trouver dans un texte et que l'homme comprend habituellement grâce à sa connaissance du domaine ? Essayons par exemple d'appliquer cette technique sur le document « *Control of the initiation of homeotic gene expression by the gap genes giant and tailless in Drosophila* » (Annexe A.1). Le résultat produit est disponible dans l'Annexe C.

Nous pouvons déjà constater que cette façon de procéder est difficilement automatisable et donc très lourde lorsqu'il s'agit d'annoter un corpus suffisant de documents. De plus, le résultat montré ci-dessus ne représente qu'une tentative de formalisation exhaustive du document et celle-ci est loin d'être parfaite. Ici, nous nous sommes tenus stricto sensu au contenu fidèle du texte sans essayer d'interpréter déjà ce que nous essayons de formaliser. Ainsi, l'auteur assimile sans doute « *les membres de la classe* ***gap*** » aux « *gènes* ***gap*** », mais nous n'avons pas représenté cette

information implicite de telle sorte que, pour la machine, ce sont deux concepts différents. Nous n'avons pas non plus représenté la notion de temps dans les verbes utilisés, qui, pour certains textes, peut se révéler d'une grande importance. Enfin, l'auteur précise clairement que la dernière assertion de ce texte est une particularité de l'assertion précédente, mais cette information n'a pas non plus été représentée.

Cet exemple illustre bien la difficulté à vouloir représenter toute la connaissance contenue dans un document, essentiellement à cause d'informations implicites que l'homme, à la différence d'une machine, peut comprendre grâce à son vécu et sa connaissance générale. Cet exercice, par son manque de rigueur, ne nous prémuni donc pas de ne pas pouvoir répondre à certaines requêtes. Mais à défaut de vouloir tout représenter, on peut se concentrer sur certains éléments en fonction du type de requêtes que l'on va poser. En réalité, cet exercice montre surtout que le choix des éléments à annoter est complètement dépendant de l'utilisation que l'on va faire de la formalisation obtenue, c'est-à-dire du type d'application qui va en tirer partie.

D'autres alternatives

On peut imaginer d'autres alternatives à une représentation exhaustive du document. D'une façon générale, on peut instaurer une certaine granularité dans la description d'un document. Prenons un exemple toujours sur le même article fournie en Annexe A.1 :

- un premier niveau de description du document pourrait faire allusion à la présence de gènes appartenant à la drosophile ;
- un second niveau préciserait l'information en disant que ces gènes appartiennent aux classes `gap` et `pair-rule` ;
- un troisième niveau identifierait précisément ces gènes en indiquant que le représentant de la classe `gap` est le gène `tailless` symbolisé par `tll` et que celui de la classe `pair-rule` est en réalité le gène `fushi-tarazu` symbolisé par `ftz` ;
- un quatrième niveau nous donnerait une information supplémentaire en signalant que ces deux gènes parfaitement identifiés ont une influence pendant le processus de segmentation de la drosophile ;
- un cinquième niveau raffinerait encore cette information en précisant que c'est en fait le gène `tailless` qui inhibe le gène `fushi-tarazu` et cela dans la partie antérieure de la mouche.

L'exemple qui suit montre concrêtement comment une interaction va être formalisée dans le format pivot ESCRIRE :

```
<esc:relation type="interaction">
   <esc:role name="promoter">
      <esc:objref id="tll"/>
   </esc:role>
   <esc:role name="target">
      <esc:objref id="ftz"/>
   </esc:role>
   <esc:attribute name="effect">inhibition</esc:attribute>
   <esc:attribute name="location">anterodorsal</esc:attribute>
</esc:relation>
```

Dépendament de l'application qui va tirer partie de cette formalisation, nous avons donc intérêt à représenter la connaissance contenue dans les documents plus ou moins finement. Dans notre cas, nous avons décidé de nous intéresser aux interactions géniques pendant le processus de segmentation de la mouche ainsi qu'à certains attributs attachés à ces interactions tels que l'effet, le moment ou le lieu de l'influence quand ces informations sont disponibles. En revanche, les conditions des expériences qui ont permis de les mettre en avant sont rarement indiquées et ne seront donc pas représentées. On voit ainsi que d'emblée, l'application va être limitée puisqu'elle ne pourra répondre qu'à un certain nombre de questions bien précises.

3.2.4 Description d'une ontologie

Nous avons vu qu'essayer de représenter formellement l'ensemble du contenu d'un document sans perdre d'informations, est une tâche extrèmement difficile. C'est pourquoi nous avons pris le parti de représenter, pour chacun des articles de notre corpus, les interactions entre gènes ou classes de gènes en plus de méta-informations classiques concernant le document. Les entités qui sont en jeu sont donc des classes de gènes, des gènes et des interactions entre eux.

Nous avons vu aussi que toutes les inférences seront locales, c'est-à-dire que seules les informations du document en cours d'évaluation seront prises en compte. Cependant, les gènes et leur organisation en classes qui sont explicitement nommés dans les différents articles, représentent bien à chaque fois les mêmes objets. Ces entités font donc partie de la description d'un monde universel et les documents ne font que des références à ces entités. L'objectif d'une ontologie est justement de décrire ce monde commun aux documents de notre corpus. C'est pourquoi, nous avons décidé de représenter les gènes, les classes de gènes et leur organisation taxonomique dans l'ontologie car ces entités servent de référence aux diverses assertions des auteurs dans les documents. La Figure 3.1 nous montre la classification des gènes qu'on peut trouver dans notre corpus de documents.

En revanche, l'influence que peut avoir tel ou tel gènes pendant le processus de segmentation de la mouche n'engage que l'auteur de l'article. Ces relations entre les objets définis dans l'ontologie seront décrites dans chacun des documents de notre corpus.

3.3 A quels types de requêtes désire-t-on répondre ?

Un des mécanismes d'inférence les plus utilisés dans une représentation de connaissances à objets concerne la structure hiérarchique des classes (taxonomie). Ainsi, dans notre contexte d'application, on pourrait imaginer la requête suivante :

> *Quels sont les documents qui traitent d'interactions entre les classes de gènes* `gap` *et* `pair-rule`*?*

Cette requête devrait retourner non seulement les documents où une interaction entre ces deux classes de gènes est explicitement spécifiée, mais aussi tous ceux qui font allusion par exemple à une interraction (ou une inhibition) entre les gènes `giant` et `fushi-tarazu`. En toute généralité, les documents pertinents devraient être ceux qui mentionnent une relation (sous-classe de `interaction`) entre une instance de la classe `gap` et une instance de la classe `pair-rule`.

3.3.1 Interprétation des requêtes

Un premier principe est que les requêtes, même si elles portent sur des entités quelconques, sont destinées à retourner des documents : ceux qui parlent de ces entités. La force des systèmes

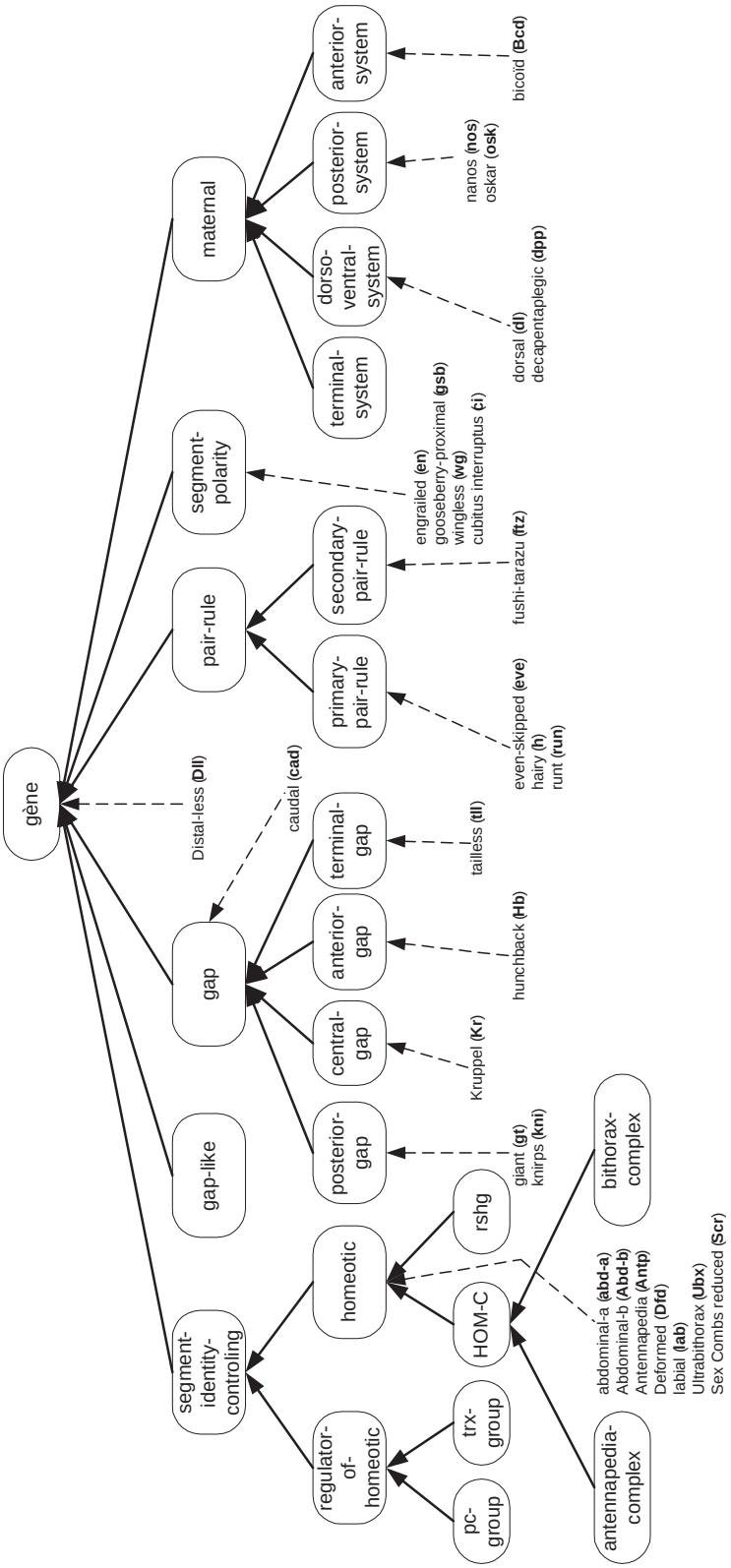

FIG. 3.1 – *Organisation taxonomique des gènes trouvés dans le corpus de documents*

à base de connaissance consiste à faire des inférences pour coller à la sémantique des langages [ESC00]. Il est donc possible d'imaginer répondre aux requêtes en faisant des inférences uniquement en fonction du contenu d'un document, ou alors en fonction du contenu de tous les documents connus. Pour mieux illustrer notre propos, nous allons développé l'exemple suivant qui décrit les deux approches qui s'offrent à nous :

- *Un document* `A` *traite d'une interaction entre le gène* `tailless` *et le gène* `hunchback`.
- *Un document* `B` *traite d'une interaction entre le gène* `giant` *et le gène* `hunchback`.
- *Un utilisateur a formulé la requête destinée à retrouver les documents traitant des interactions communes aux gènes* `tailless` *et* `giant`.

La première solution consiste en fait à former une base de connaissances dont la matière première serait l'ensemble des annotations contenues dans les documents. Ainsi, chaque objet de chaque document du corpus est extrait pour former une certaine terminologie. Ensuite, on est capable d'indiquer pour chaque document du corpus sa pertinence relative à l'ensemble des termes de la terminologie formée. L'inconvénient de cette approche classique, c'est qu'à chaque nouveau document annoté, la terminologie est susceptible de grossir et qu'il faudrait donc théoriquement reparcourir tous les documents pour les re-indexer. Dans notre contexte spécifique d'application, cela n'est pas utile puisqu'on peut affirmer que les documents déjà indexés ne seront pas pertinents pour chaque nouveau terme venant s'ajouter à la terminologie existante. Cette approche va nous permettre d'effectuer ce qu'on pourrait appeler des *inférences globales*. En effet, les connaissances contenues à l'intérieur de chaque document sont regroupées. Ainsi, lorsqu'une requête est formulée, le moteur peut s'aider de toute cette connaissance pour effectuer des inférences. Le résultat de la requête indiquée dans l'exemple précédent contiendra donc les documents `A` et `B` car le gène `hunchback` interagit communément avec les gènes `tailless` et `giant`.

Mais indexer les documents par le contenu n'est pas la même chose que construire une base de connaissances. En effet, le but de notre application n'est pas de pouvoir répondre à n'importe quelles questions relatives à la drosophile, mais bien de retourner un groupe de documents permettant de répondre par l'affirmative à une requête. L'autre solution consiste donc simplement à tenir à jour un index des pages annotées. Ainsi, lorsqu'une requête est formulée, le moteur de recherche va charger une à une les pages annotées et décider si le document est pertinent ou non pour la requête. Cette approche a le mérite de se baser uniquement sur la représentation du contenu que l'on a intégré (et de l'ontologie) pour juger de la pertinence d'un document. Il fait donc totalement abstraction des annotations figurant dans les autres documents du corpus. Ainsi, le résultat de la requête formulée dans l'exemple précédent *NE* contiendra *PAS* les documents `A` et `B`. En effet, chacun de ces documents pris indépendament ne permet pas d'affirmer que le gène `hunchback` interragit avec les gènes `tailless` et `giant`. Cette dernière solution, qui ne consiste à faire que des *inférences locales*, a donc finalement été adoptée.

3.3.2 Langages de requêtes

Une fois les documents annotés, il faut encore fournir à un utilisateur la possibilité de formuler des requêtes. Le langage de requête peut reprendre le schéma classique introduit par SQL :

- *Présentation* (`SELECT`) : comme nous l'avons déjà dit, l'objectif des requêtes est de retourner des documents. L'aspect présentation n'étant donc pas le but de notre application, il se réduira à la liste des documents pertinents avec éventuellement la valeur des autres paramètres.

- *Typage* (`FROM`) : cette clause sert à typer et introduire les variables.
- *Sélection* (`WHERE`) : la partie sélection est la partie requête proprement dite. C'est là qu'est en grande partie la difficulté et on va y revenir ci-dessous.
- *Pertinence* (`ORDER BY`) : cette clause sert à ordonner les résultats par importance, celle-ci pouvant être exprimée en fonction de l'importance des contraintes à satisfaire, de la taille du document, de la proximité de celui-ci par rapport à la requête...

Ainsi, la requête de l'exemple donné précédemment :

Quels sont les documents qui traitent d'interractions entre les classes de gènes `gap` *et* `pair-rule` *?*

pourrait être exprimée comme suit :

```
SELECT
FROM interaction:I
WHERE I.promoter-class = "gap"
  AND I.target-class = "pair-rule"
```

ou alors comme suit :

```
SELECT
FROM interaction:I
WHERE I.promoter ∈ "gap"
  AND I.target ∈ "pair-rule"
```

La clause `WHERE` est la partie la plus complexe de la requête. Elle doit spécifier les filtres à mettre en place pour isoler les documents pertinents. Ces filtres concernent les sélections possibles sur les attributs des objets et sur leurs valeurs. Ainsi, le langage de requête doit pouvoir effectuer une sélection sur les valeurs des attributs, mais aussi sur la classe à laquelle appartiennent les individus. Il doit enfin pouvoir combiner les modes de sélection. Nous revenons sur chacun de ces aspects.

La sélection sur les valeurs des attributs peut être raffinée par le mode de comparaison des valeurs, à savoir :

- l'*égalité* : la valeur de l'attribut doit être égale à une valeur particulière ;
- l'*énumération* : la valeur de l'attribut doit être dans un ensemble de valeurs particulières ;
- l'*intervalle* : la valeur de l'attribut doit être dans un intervalle de valeurs particulier ;
- la *classe* : la valeur de l'attribut doit être dans une classe d'objets particulière ;

Il semble intéressant de pouvoir disposer de chacun de ces modes de comparaison. La sélection des attributs peut être, elle, raffinée sur le mode de l'ODMG. On peut ainsi exprimer un chemin grâce au `'.'` entre les différents attributs. Enfin, on peut s'interroger sur l'utilité de quantificateurs universel ou existentiel. En effet, faut-il interpréter une requête comme, il existe un objet dans le document qui la satisfait, ou tous les objets dans le document doivent la satisfaire ? Ces deux aspects nous semblent intéressants et seront donc marqués explicitement par un quantificateur.

La sélection sur la(les) classe(s) d'appartenance consiste à déterminer si un individu est attaché à une classe particulière. Enfin, la sélection combinée peut aisément être réalisée grâce aux opérateurs booléens `AND`, `OR` et `NOT`.

Le Tableau 3.1 résume les différentes contraintes évoquées en donnant la grammaire sous forme EBNF de la clause `WHERE`.

where	::=	contrainte (op contrainte)*	// *sélection*
op	::=	`AND` | `OR`	// *connecteurs logiques*
contrainte	::=	`NOT` contrainte | quant comp	// *contrainte sur attribut*
quant	::=	`ALL` | `EXIST`	// *quantificateurs*
comp	::=	terme predicat terme	// *comparaison*
terme	::=	path | const	// *terme*
predicat	::=	$= \mid \leq \mid \in$	// *prédicat*
path	::=	attname (. attname)*	// *chemin*
const	::=	integer | string | set | list	// *valeur*

TAB. 3.1 – *Grammaire sous forme EBNF de la clause* `WHERE`

3.4 Comment intégrer cette représentation formelle au document?

Une dernière question concerne le stockage physique des annotations relatives à un document. La cible visée est le format HTML ou plutôt XHTML car une représentation du contenu dans un format XML est ajoutée. Une idée simpliste consiste donc à intégrer ces annotations dans le document lui-même. La seule condition est d'interdire qu'un navigateur affiche ces méta-données. Pour cela, ces annotations doivent être placées dans l'en-tête du document et être encapsulées dans des balises RDF puisque tout le contenu de celles-ci est ignoré par les navigateurs actuels. Cette solution présente un certain avantage puisque le contenu et les annotations se retrouvent dans le même fichier et l'annotateur peut donc travailler sans être connecté à un site distant. Elle a d'ailleurs été retenue.

Cependant, on peut suggérer une autre solution consistant à séparer clairement le document, de ses annotations. Cette solution procure en effet un autre avantage : n'importe qui peut annoter des pages sans être propriétaire du document. De plus, plusieurs types d'annotations concernant le même document peuvent coexister, et donc être utilisés différement selon le type d'usager. Un serveur de base de données permet alors de faire la correspondance entre les documents (distants), et les annotations (locales). Si le fait d'être propriétaire du document ne semble pas être un argument décisif dans notre contexte, la possibilité d'annoter un document à plusieurs et de manière concurente est en revanche très intéressante. Ce type d'application est actuellement en cours de développement au W3C et son utilisation est donc envisageable dans le futur.

3.5 Conclusion

L'étude menée dans ce chapitre a tenté de montrer les questions que l'on doit se poser pour pouvoir formaliser la connaissance contenue dans des documents et produire ensuite le système qui va en tirer partie. Nous avons proposé un certain nombre de réponses et nous pouvons en tirer une première conclusion : le choix des éléments à annoter dépend de l'objectif poursuivi, c'est-à-dire de l'application qui va utiliser ces annotations.

Nous disposons d'un corpus de documents relatifs aux interactions géniques ayant lieu pendant le processus de segmentation de la drosophile. Ces documents sont en réalité des résumés d'articles scientifiques. Nous avons décidé de nous intéresser principalement à la nature de ces interactions ainsi qu'aux gènes et aux classes concernés. Nous avons pris le parti de séparer l'ontologie (c'est-à-dire la description des gènes et des classes de gènes), des individus (c'est-à-dire la déclaration des interactions entre gènes). Ainsi, chaque document fait référence à l'ontologie et possède un certain nombre d'objets et de relations entre eux. Par contre, un document ne

peut pas déclarer d'élément conceptuel, c'est-à-dire compléter l'ontologie. Dans notre contexte, cette contrainte n'est pas si gênante puisque l'ontologie peut dès le départ être décrite d'une façon relativement stable. Toutefois, il faut garder à l'esprit qu'un nouveau concept d'interaction pourrait apparaître remettant ainsi en cause le schéma défini. Nous allons maintenant étudier les moyens à mettre en œuvre pour implémenter les solutions proposées.

Chapitre 4

Mise en œuvre

Nous avons vu dans le chapitre précédent quels types d'éléments nous allons annoter. Notre corpus de documents concerne les interactions géniques pendant le processus de segmentation chez la drosophile. Nous avons considéré que les gènes et leur organisation taxonomique en classes font partie d'une description commune à tous les documents du corpus et appartiennent donc à l'ontologie. Les interactions entre les gènes sont elles, représentées à l'intérieur de chaque document selon les assertions de leurs auteurs.

Nous allons maintenant décrire la mise en œuvre des solutions proposées. Nous commencerons donc par présenter au début du chapitre le système TROEPS, un représentant des systèmes de représentation de connaissances à objets. Nous indiquerons en particulier les particularités de ce système puisqu'il va héberger notre base de connaissance (section 4.1). Nous reviendrons alors sur l'annotation des documents proprement dit, en insistant sur les différentes étapes composant le processus et sur leur implémentation (section 4.2). Nous étudierons ensuite comment l'utilisateur pourra interroger notre corpus de documents annotés. En particulier, nous verrons quel type d'interface permettra de composer des requêtes et comment celles-ci seront évaluées (section 4.3).

4.1 Présentation de TROEPS

Le système de gestion de bases de connaissances à objets TROEPS [SHE95] a été développé dans l'équipe SHERPA, à l'INRIA Rhône-Alpes. TROEPS est un représentant de la représentation de connaissances à objets. Une de ses particularités est de séparer les connaissances selon leur évolutivité. En effet, alors qu'une partie des caractéristiques d'une entité reste immuable, d'autres peuvent changer traduisant le changement dans le monde ou dans la vision du monde [Val99]. Ainsi, TROEPS partage délibérément les connaissances génériques entre le concept et la classe.

Un objet est donc instance d'un seul concept, mais peut changer de classe, voire en avoir plusieurs simultanément. Les descriptions des classes en TROEPS ne sont que des conditions nécessaires (mais non suffisantes) pour l'appartenance d'un objet à la classe. Dans la suite, nous allons décrire le modèle d'objet dans TROEPS, et plus spécifiquement pour l'ontologie construite autour de notre corpus de documents.

Le concept et ses attributs

Les objets d'une base de conaissances TROEPS sont partitionnés en un ensemble de *concepts*. Chaque concept représente une famille d'individus du monde modélisé qui partagent l'ensemble

de leurs caractéristiques (exprimées moyennant les attributs des objets). Ainsi, le concept défini la structure et l'identité de ses instances par la donnée de l'ensemble de leurs attributs et des types de ces attributs. Un objet appartient à exactement un concept de la base. En effet, au sein du concept, tout nom est unique et deux entités portant le même nom sont considérées comme identiques. L'ontologie issue de notre corpus de documents contient des concepts tels que `gene` ou `interaction`.

Un concept fixe l'ensemble des attributs qui décrivent les instances du concept. Chaque attribut défini par le concept s'applique à toutes les instances de celui-ci. Ainsi, l'attribut peut être vu comme une fonction qui associe à tout objet une (ou un ensemble de) valeur(s). Pour un attribut donné, sa description au sein du concept contient des informations précisant son type, sa nature et son constructeur qui sont mises à profit lors de diverses opérations de manipulation des objets telles que le calcul de la spécialisation et la classification d'objets et de classes. Le *type* d'un attribut décrit l'ensemble des valeurs possibles que l'attribut peut prendre dans les instances du concept, c'est-à-dire le co-domaine de la fonction sous-jacente. Ce type peut être soit un concept TROEPS, soit un type de données simple. Mais pour rendre extensible la gestion de types, TROEPS ne contient pas de types primitifs et tout type de données est donc importé de l'extérieur. La *nature* d'un attribut offre des précisions d'ordre sémantique sur la manière dont les valeurs de l'attribut caractérisent les objets sous-jacents. Ainsi, un attribut pourra être de nature `propriété` et sa valeur sera de type simple, ou de nature `lien` caractérisant une relation entre des individus du monde modélisé. Le type est alors un concept et la valeur, un objet. Enfin, le *constructeur* d'un attribut indique le nombre de valeurs possibles que l'attribut peut prendre dans un objet. Les constructeurs possibles sont donc `un`, mais aussi `liste` ou `ensemble`.

Par exemple, le concept `interaction` possède des attributs correspondant à la nature (`type`), le moment (`start-minutes`, `end-minutes`) ou la localisation (`location`) de l'interaction, mais aussi aux gènes ou aux classes de gènes impliqués en tant que cible ou instigateur de celle-ci (`target-class`, `effector-class`). Les attributs `target-class` ou `effector-class` de ce concept sont de nature `propriété` et l'attribut correspondant aux méthodes expérimentales ayant mises en évidence cette interaction (`experimental-methods`) a comme constructeur un `ensemble`.

Le point de vue de concept

Un concept n'est manipulé que sous une certaine perspective qui fait référence à un sous-ensemble de ses caractéristiques. Une perspective correspond à la vision particulière qu'un spécialiste du domaine modélisé détient sur celui-ci [DEMN98]. Dans notre contexte, la perspective définit un ensemble d'attributs visibles des objets du concept, ainsi qu'une manière spécifique de les organiser en classes. Ainsi, un concept se voit associer un ensemble de *points de vue*, chacun muni de sa propre taxonomie de classes. Par exemple, le concept `interaction` peut être vu plutôt selon la structure de l'instigateur ou selon celle de la cible ou encore selon le résultat que cette influence produit (Figure 4.1).

La classe racine de la taxonomie contient la totalité des objets du concept et est progressivement spécialisée en sous-classes plus restreintes. Ainsi, la classe racine du concept `interaction` sera toujours le concept lui-même (`interaction`). Celle-ci est ensuite dérivée en plusieurs classes. Par exemple, dans le point de vue `result`, la classe `interaction` est spécialisée en trois sous-classes : `repressor-and-activator`, `repressor` et `activator` (Figure 4.1).

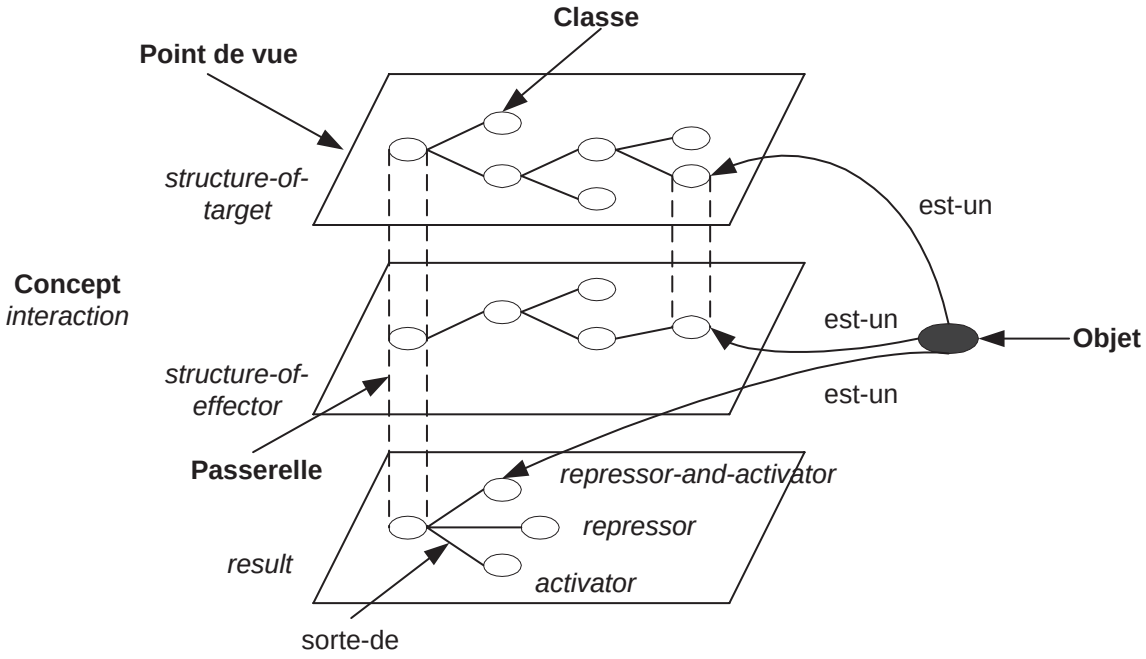

FIG. 4.1 – *Exemple de trois points de vue sur le concept* `interaction`*:* `structure-of-target`*,* `structure-of-effector` *et* `result`

La classe et ses attributs

Les *classes* TROEPS décrivent des sous-ensembles significatifs d'instances d'un concept. Ainsi, une classe est associée à un concept TROEPS unique. Les classes sont aussi associées à un point de vue unique sur le concept sous-jacent. Des passerelles permettent d'établir des liens entre les classes appartenant à des points de vue différents sur le même concept. Celles-ci sont composées par un ensemble de classes de départ et une classe d'arrivée. Leur signification est la suivante : tout objet qui appartient à l'ensemble des classes de départ, appartient nécessairement à la classe d'arrivée.

Les attributs de classe constituent un sous-ensemble des attributs du concept sous-jacent mais possèdent une structure propre. Ainsi, l'attribut définit des restrictions sur les valeurs de l'attribut de même nom dans les objets membres de la classe. Chaque restriction s'interprète comme un ensemble de valeurs admissibles appartenant au type de l'attribut, défini dans le concept.

Les objets

Un *objet* TROEPS peut être vu comme un ensemble de couples <attribut, valeur>muni d'une identité. Cette identité est établie en TROEPS grâce à une clé. L'ensemble des attributs possédés par l'objet dont la clé constitue un sous-ensemble, est défini au sein de l'unique concept dont l'objet est instance. Les valeurs des attributs clés permettent l'identification de l'objet à l'intérieur du concept. Un objet est rattaché à une seule classe plus spécifique sous chaque point de vue de son concept. L'objet est membre de (appartient à) ces classes, ainsi que toutes leurs super-classes. Pour chaque classe d'appartenance, les valeurs des attributs de l'objet appartiennent aux plages spécifiées par les attributs de la classe. Ainsi, un objet TROEPS instance du concept `interaction`

sera par exemple « `interaction "TLL_hb"` » indiquant une influence du gène `tailless` sur le gène `hunchback`. Finalement la Figure 4.1 résume les principaux points abordés en montrant le concept `interaction` sous trois points de vue. Une instance de ce concept est visualisée avec ses liens d'attachement sous chaque point de vue.

4.2 Annotation des documents

Comme nous l'avons déjà signalé, notre corpus de documents provient de la base Medline. Une simple requête sur leur moteur de recherche permet de récupérer un ensemble de résumés d'articles scientifiques, correspondant à la requête saisie, dans une page HTML (Annexe A.1). La base fournit en outre un ensemble de méta-données pour chacun de ces articles (auteur(s), titre, langue, journal, pages, date de publication, date d'indexation dans la base ...) dans un format parfaitement spécifié (Annexe A.2). Cette base de documents constitue le support sur lequel nous allons ajouter des annotations.

Nous allons donc détailler les différentes étapes qui permettent d'annoter les documents (section 4.2.1) avant d'aborder l'implémentation proprement dîte (section 4.2.2). Nous donnerons finalement un exemple de document annoté sur le résumé de l'article fourni en Annexe A.1 (section 4.2.3).

4.2.1 Les différentes étapes de l'annotation

La Figure 4.2 illustre le processus, qui, à partir des documents fournis par Medline, en produit de nouveaux complètement annotés. Nous allons revenir sur chacune des étapes clés.

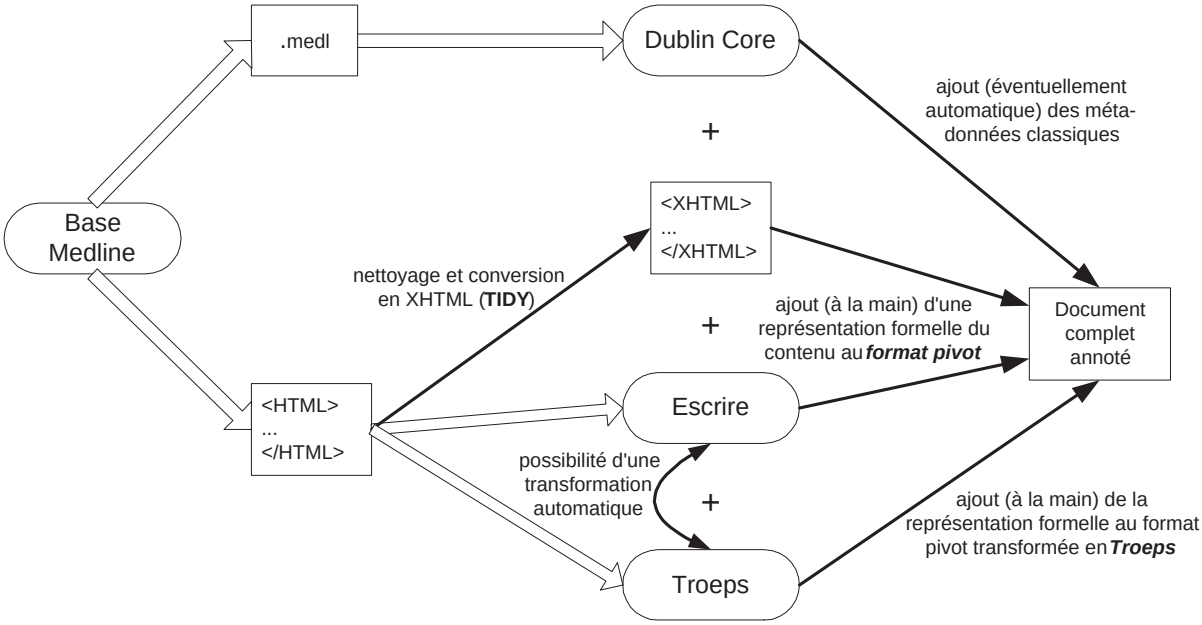

FIG. 4.2 – *Déroulement du processus d'annotation d'un document*

Conversion en XHTML des fichiers Medline

Les résumés fournis par Medline sont au format HTML. Mais l'ensemble des annotations qui vont être rajoutées sont produites au format XML. Il est donc nécessaire de bien former les éléments HTML (au sens de bien-formé en XML) des fichiers de la base Medline. Le W3C a normalisé le langage permettant d'obtenir ce type de document. XHTML 1.0 ([RLJ00]) est ainsi définit comme une reformulation de HTML 4 ([ACC+97]) en une application XML 1.0 ([BPS98]), et trois DTDs correspondant à celles définies par HTML 4. Les principales modifications à effectuer pour convertir du HTML en XHTML sont donc de vérifier que tous les éléments soient fermés et que tous les attributs soient « *quotés* ».

Tidy [2] est un programme conçu par Dave Raggett permettant justement de transformer des documents HTML au format XHTML. Il analyse ainsi le fichier HTML donné en entrée et essaie d'ajouter les éléments manquants pour obtenir un arbre XML.

Intégration d'une représentation formelle du document au format pivot

L'essentiel de notre étude porte sur l'intégration d'une représentation formelle de la connaissance contenue dans un document. Comme nous l'avons déjà signalé, cette étude est réalisée dans le cadre de l'action ESCRIRE qui a pour but de comparer trois logiciels utilisant des systèmes de représentation de connaissances différents (graphes conceptuels, représentation de connaissances à objets et logiques de descriptions). Mais pour que les mesures soient objectives, il est nécessaire que les entrées/sorties des logiciels soient les mêmes, d'où l'utilisation d'une structure pivot.

Nous avons vu dans le chapitre précédent comment annoter les documents de notre corpus en précisant notamment quel type d'élément nous allons représenter. Ceci nous a permis de définir une ontologie de notre domaine d'application. La DTD du format pivot ESCRIRE (Annexe B.1) contraint ainsi les classes, objets et relations susceptibles d'apparaître dans chaque document puisque les annotations intégrées dans ceux-ci seront valides vis à vis de cette DTD.

Intégration de cette même représentation au format TROEPS

TROEPS est le système qui va gérer les connaissances contenues dans les documents. C'est en fait un langage de représentation de connaissances à objets qui permet d'éditer de nouvelles bases de connaissances (voir section 4.1). Il bénéficie pour cela d'une interface XML spécifiée dans une DTD. On peut donc mettre en œuvre une traduction automatique entre le format pivot XML défini dans l'Annexe B.1 et le format d'entrée TROEPS.

Intégration de méta-données classiques

En plus d'une représentation formelle du contenu, il paraît intéressant d'ajouter à chacun des documents des méta-données plus «classiques» concernant par exemple l'auteur, le titre ou la langue du document. Le *Dublin Core Metadata Initiative* [3] est justement un ensemble de 15 «méta-attributs» (c'est-à-dire de descripteurs de méta-données) destiné à indexer n'importe quel document sur le réseau. Ces descripteurs comprennent des éléments de signalisation classiques (titre, auteur, version, éditeur, langue...) mais aussi des données plus spécifiques à l'environnement informatique (le type, le format ou l'identifiant d'un document). Il est largement fondé sur les standards de l'internet (URI pour l'identifiant d'un document, MIME pour son type ou la

2. http://www.w3.org/People/Raggett/tidy/
3. http://www.purl.com/dc/

norme ISO639 pour représenter le code de sa langue). Traduite dans de nombreux langages, cette proposition est à la fois ouverte et consensuelle.

Medline fournit pour chacun de ses documents quasiment toutes ces informations dans un format parfaitement spécifié (représenté par un fichier séparé `.medl` dans la Figure 4.2). Un exemple est d'ailleurs donné dans l'Annexe A.2. Il est donc aisé d'extraire automatiquement ces informations et de les convertir au format Dublin Core pour les intégrer à notre document.

4.2.2 Implémentation

Le processus d'annotation des documents met en avant le besoin d'automatiser certaines tâches. L'interface du système avec l'utilisateur se fera par l'intermédiaire du Web et donc du langage HTML. Mais les services proposés doivent être plus complexes que le simple envoi d'un document HTML figé. Le serveur Web doit en effet être capable de générer des réponses dynamiquement. Ces programmes qui vont générer les pages HTML peuvent être implémentés sous la forme de scripts CGI (*Common Gateway Interface*) ou alors de *servlets*, l'équivalent en JAVA. L'Annexe D décrit l'architecture et la programmation du serveur de servlets utilisé. Cette présentation est inspirée de [Boy99].

Deux méthodes HTTP permettent de transmettre les informations au serveur : GET et PUT. PUT permet justement d'envoyer des informations. En ce qui concerne notre application, les paramètres sont transportés dans l'URL et chaque servlet est donc chargé de répondre à une requête de type PUT (implémentation de la méthode *doPost*). Un formulaire [4] permet de donner les paramètres de ces servlets (Figure 4.3). Le serveur Web *Apache* [5] (et son module *JServ*) est chargé d'exécuter les servlets. Le processus d'annotation de documents contient deux types de servlets :

- Le servlet `TidyServlet` permet de convertir un fichier HTML en XHTML. Basé sur le programme *Tidy* développé par Dave Ragget, il prend en paramètre l'URL d'un fichier HTML. Il essaie alors de le convertir et retourne les avertissements et erreurs rencontrés lors de la transformation. Si un fichier XHTML a pu être généré, un lien pointe vers ce fichier résultat.

- Le servlet `XTServlet` sert lui, à appliquer une feuille de style XSL sur un document XML (ou XHTML). Il est basé sur XT [6], une implémentation de XSLT réalisée par James Clark. Il prend donc en paramètre l'URL d'un fichier source XML, ainsi que l'URL d'une feuille de style XSL. Il analyse alors le fichier source, lui applique la feuille de style et produit un fichier contenant le résultat. Deux feuilles de style sont disponibles permettant respectivement d'extraire les annotations au format ESCRIRE et au format TROEPS d'un fichier XHTML. Cet servlet permet donc d'isoler les annotations ajoutées dans les deux langages.

4.2.3 Exemple de document annoté

Nous avions déjà essayé de représenter le contenu de l'article fourni en Annexe A.1. Voici le résultat produit pour ce même article dans le format pivot ESCRIRE.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE esc:content SYSTEM "http://escrire.inrialpes.fr/dtd/escrire.dtd">
```

4. http://escrire.inrialpes.fr/transformation.html
5. http://java.apache.org
6. http://www.jclark.com/xml/xt.html

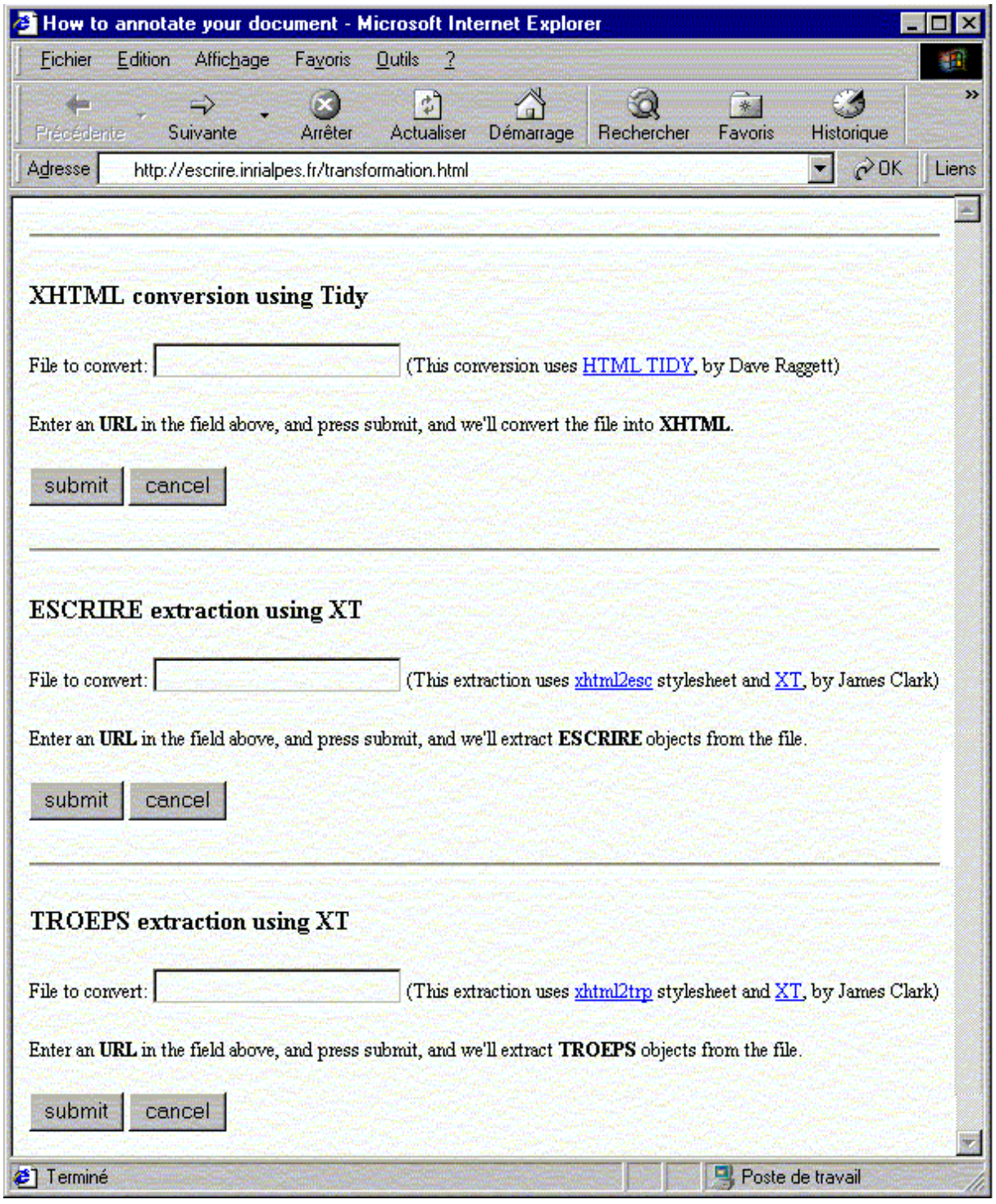

FIG. 4.3 – *Formulaire permettant d'accéder aux servlets implémentés*

```
<esc:content ontology="http://escrire.inrialpes.fr/biointer/biointer.xml"
            url="http://escrire.inrialpes.fr/biointer/esc/1972684.xml">
   <esc:objref id="gap"/>
   <esc:objref id="pair-rule"/>
   <esc:objref id="homeotic"/>
   <esc:objref id="tll"/>
   <esc:objref id="gt"/>
   <esc:objref id="Abd-B"/>
   <esc:objref id="Antp"/>
   <esc:objref id="hb"/>
   <esc:objref id="ftz"/>
   <esc:objref id="Dfd"/>
   <esc:relation type="interaction">
      <esc:role name="promoter-class">
         <esc:objref id="gap"/>
      </esc:role>
      <esc:role name="target-class">
         <esc:objref id="pair-rule"/>
      </esc:role>
   </esc:relation>
   <esc:relation type="interaction">
      <esc:role name="promoter-class">
         <esc:objref id="gap"/>
      </esc:role>
      <esc:role name="target-class">
         <esc:objref id="homeotic"/>
      </esc:role>
   </esc:relation>
   <esc:relation type="interaction">
      <esc:role name="promoter">
         <esc:objref id="tll"/>
      </esc:role>
      <esc:role name="target">
         <esc:objref id="ftz"/>
      </esc:role>
      <esc:attribute name="effect">inhibition</esc:attribute>
      <esc:attribute name="location">anterodorsal</esc:attribute>
   </esc:relation>
   <esc:relation type="interaction">
      <esc:role name="promoter">
         <esc:objref id="gt"/>
      </esc:role>
      <esc:role name="target">
         <esc:objref id="Abd-B"/>
      </esc:role>
   </esc:relation>
   <esc:relation type="interaction">
      <esc:role name="promoter">
         <esc:objref id="gt"/>
      </esc:role>
      <esc:role name="target">
         <esc:objref id="Antp"/>
      </esc:role>
      </esc:relation>
   <esc:relation type="interaction">
      <esc:role name="promoter">
         <esc:objref id="tll"/>
      </esc:role>
      <esc:role name="target">
```

```
            <esc:objref id="hb"/>
         </esc:role>
         <esc:attribute name="effect">inhibition</esc:attribute>
         <esc:attribute name="location">anterodorsal</esc:attribute>
      </esc:relation>
      <esc:relation type="interaction">
         <esc:role name="promoter">
            <esc:objref id="tll"/>
         </esc:role>
         <esc:role name="target">
            <esc:objref id="Dfd"/>
         </esc:role>
         <esc:attribute name="effect">inhibition</esc:attribute>
         <esc:attribute name="location">anterodorsal</esc:attribute>
      </esc:relation>
      <esc:relation type="interaction">
         <esc:role name="promoter">
            <esc:objref id="tll"/>
         </esc:role>
         <esc:role name="target">
            <esc:objref id="Abd-B"/>
         </esc:role>
         <esc:attribute name="effect">activation</esc:attribute>
         <esc:attribute name="location">posterior</esc:attribute>
      </esc:relation>
      <esc:relation type="interaction">
         <esc:role name="promoter">
            <esc:objref id="gt"/>
         </esc:role>
         <esc:role name="target">
            <esc:objref id="hb"/>
         </esc:role>
         <esc:attribute name="effect">activation</esc:attribute>
         <esc:attribute name="location">anterior</esc:attribute>
      </esc:relation>
   </esc:content>
```

4.3 Interrogation des documents

Nous venons de voir le processus mis en œuvre pour annoter les documents de notre corpus. Nous avons aussi présenté le système TROEPS qui va gérer toutes les connaissances que nous avons représenté. C'est notamment lui qui va effectuer des inférences sur ces connaissances pour évaluer des requêtes. Une fois les documents annotés, il nous reste donc à fournir à un utilisateur la possibilité de formuler des requêtes.

Nous allons donc présenté les différentes étapes permettant d'interroger notre corpus de documents (section 4.3.1) avant d'aborder l'implémentation mise en œuvre (section 4.3.2). Nous donnerons alors un exemple de requête en XML avec le résultat de son évaluation (section 4.3.3).

4.3.1 Les différentes étapes de l'interrogation

Le processus permettant de composer, puis d'évaluer des requêtes avant de retourner une réponse est illustré dans la Figure 4.4 et détaillé dans la suite.

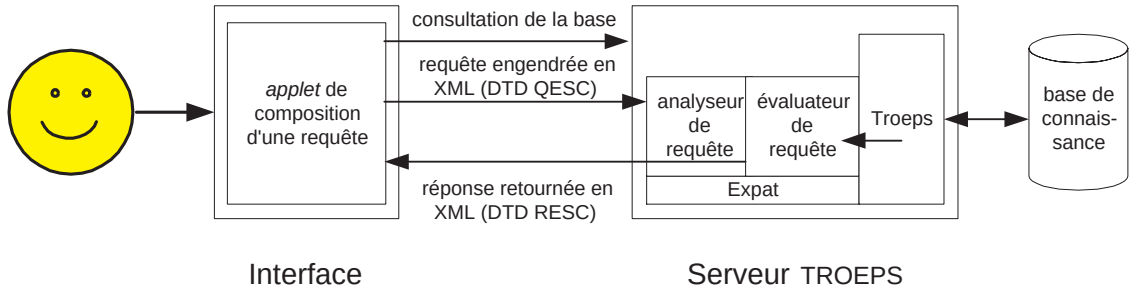

FIG. 4.4 – *Composition et évaluation d'une requête*

Composition de la requête

L'utilisateur va pouvoir composer des requêtes via une *applet* dans une page HTML. L'interface proposée va permettre de construire graphiquement ces requêtes. Elle est ainsi un bon compromis entre simplicité et expressivité du langage. Il faut auparavant avoir identifié précisément quel type de requête un utilisateur est susceptible de formuler. Dans notre contexte, ces requêtes concernent principalement la nature, le lieu ou le moment d'une interaction ainsi que les gènes impliqués et leur classe.

Mais cette interface peut aussi s'inspirer de ce qui se fait maintenant traditionnellement dans les moteurs de recherche sur le Web. Ainsi l'usager pourrait saisir une liste de mots clés séparés par des connecteurs logiques, ou alors naviguer à travers des catégories pré-établies et regroupant déjà les documents. Dans notre contexte, on peut imaginer des « *super-catégories* » `gènes` ou `interactions` correspondant en fait aux concepts TROEPS. Elles seraient alors divisées en sous-catégories et permettraient par exemple de retrouver l'organisation taxonomique des gènes. De même, une catégorie `interactions` serait, en fait, divisée en toutes les influences entre gènes et classes de gènes citées dans les documents de notre corpus. Cependant, cette manière d'interroger les documents sera conseillée aux novices dans le domaine car elle ne permet pas de formuler des requêtes très compliquées. En fait, l'utilisateur est déjà en difficulté s'il veut utiliser la conjonction de deux faits dans sa requête. Le mode permettant d'interroger les documents par une liste de mots clés est lui plutôt réservé à un utilisateur expérimenté. Il a l'avantage d'exploiter pleinement l'expressivité du langage de requête mais nécessite de connaître ce langage (les connecteurs logiques permettant de relier les mots clés).

Analyse de la requête engendrée

Nous avons déjà vu pourquoi nous avons utilisé une structure pivot XML pour représenter la connaissance contenue dans les document. L'action ESCRIRE ayant pour objectif la comparaison de trois formalismes, il est nécessaire que ceux-ci soient placés sur un pied d'égalité et disposent donc des mêmes entrées et sorties. C'est la raison pour laquelle les requêtes ont aussi été engendrées en XML. TROEPS dispose d'une interface XML pour construire et charger de nouvelles bases de connaissances. Les requêtes sont elles généralement produites en naviguant dans la base via une interface HTML. L'URL est alors décodée et des filtres créés pour pouvoir ramener les objets pertinents répondant à la requête. Une autre solution consiste à engendrer directement ces filtres, c'est-à-dire une structure LISP à laquelle TROEPS saura immédiatement répondre.

Réponse retournée

L'évaluation de la requête retourne des réponses en XML valides vis à vis de la DTD fournie dans l'Annexe B.3. Chacune de ces réponse est composée de valeurs pour les variables données dans la requête et d'une URL pointant sur le document jugé pertinent. Une feuille de style peut alors afficher le résultat produit dans un navigateur traditionnel.

4.3.2 Implémentation

La conception de l'interface permettant de formuler des requêtes n'étant pas terminée, des requêtes sont produites à la main pour être injectées directement dans le système TROEPS. Elles sont au format XML et valides vis à vis de la DTD fournie dans l'Annexe B.2.

L'analyse des requêtes XML fournies va permettre d'engendrer une structure LISP. *Expat* [7] est une librairie C permettant d'analyser syntaxiquement le langage XML. Conçu par James Clark, c'est un analyseur évènementiel dans le sens où il ne se contente pas de construire l'arbre XML correspondant, mais permet bien d'effectuer des tâches à chaque élément rencontré. Nous avons donc mis en œuvre et intégré à TROEPS un tel analyseur pour pouvoir engendrer directement les filtres correspondant à la requête XML. L'analyse s'effectue relativement simplement avec une pile, puisque la DTD des requêtes est parfaitement spécifiée, et retourne donc une liste d'éléments correspondant aux quatre clauses possibles d'une requête (`select`, `from`, `where`, `order by`).

Nous avons défini dans la section 3.3 la statégie utilisée pour interpréter les requêtes. Chaque document est ainsi évalué séparément, c'est-à-dire en se basant uniquement sur les connaissances du document représentées. Cela conduit donc à n'effectuer que des inférences locales par opposition à inférer de nouvelles connaissances globalement, c'est-à-dire en regroupant toutes les connaissances contenues dans les documents. On peut imaginer au moins deux stratégies mettant en œuvre cette solution (Tableau 4.1) :

- Une première stratégie consiste pour chaque document à le charger dans la base (c'est-à-dire à charger les objets correspondant aux annotations du document), à évaluer la requête, puis à décharger le document. Ainsi, chaque requête transmise sera évaluée un nombre de fois égal au nombre de documents composant notre corpus. Il n'y a pas d'indexation des documents.

- Une seconde stratégie consiste à charger tous les documents d'un seul coup dans la base. La requête est alors évaluée une seule fois et tous les documents sont indexés par rapport aux objets qu'ils contiennent. Une liste de tuples d'objets attachés à des documents est retournée et il ne reste donc plus qu'à appliquer les éventuels connecteurs logiques de la requête sur ces documents.

C'est la deuxième stratégie qui a été implémentée. L'évaluation de la requête proprement dîte se déroule comme suit :

1. on créé un filtre pour chaque variable présente dans la requête ;

2. on normalise ensuite la requête (les connecteurs existentiel $\exists$ et universel $\forall$ sont sortis à l'extérieur, le connecteur logique `NOT` est lui, rentré à l'intérieur des termes) ;

3. on applique les contraintes unaires sur les filtres ;

7. http://www.jclark.com/xml/expat.html

pour tout les documents		**pour tout** les documents
charger le document		charger le document
évaluer la requête	*vs*	**fin pour**
décharger le document		engendrer le contexte
fin pour		évaluer la requête
si tri **alors** le faire		**si** tri **alors** le faire

TAB. 4.1 – *Deux stratégies pour évaluer les requêtes*

4. on engendre alors un contexte, c'est-à-dire un tuple d'objets attachés à des documents ;
5. on applique les connecteurs logiques de la requête sur les documents retournés dans les tuples ;
6. on ordonne éventuellement les résultats.

4.3.3 Exemple de requête

Imaginons la requête suivante :

Quels sont les documents parlant d'une interaction du gène "gt" (giant) sur "Hb" (hunchback) ?

Celle-ci se traduira dans le format pivot ESCRIRE par :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE esc:query SYSTEM "http://escrire.inrialpes.fr/dtd/qesc.dtd">
<esc:query url="http://escrire.inrialpes.fr">
   <esc:select />
   <esc:from>
      <esc:relvar type="interaction" id="I" />
   </esc:from>
   <esc:where>
      <esc:and>
         <esc:eq>
            <esc:path>
               <esc:relvarref type="interaction" id="I" />
               <esc:attribute name="promoter" />
            </esc:path>
            <esc:objref id="gt" />
         </esc:eq>
         <esc:eq>
            <esc:path>
               <esc:relvarref type="interaction" id="I" />
               <esc:attribute name="target" />
            </esc:path>
            <esc:objref id="Hb" />
         </esc:eq>
      </esc:and>
   </esc:where>
</esc:query>
```

L'évaluateur de requête retourne la réponse suivante :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<esc:resc ontology="">
   <esc:answer>
      <esc:vref name="I"/>(tr-object (tr-concept "interaction") '("1972684" 9 ))
      </esc:vref>
      <esc:docref href="http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve
               &db=PubMed&dopt=Abstract&list_uids=1972684"/>
   </esc:answer>
</esc:resc>
```

4.4 Conclusion

Nous venons de présenter le système de représentation de connaissances TROEPS qui va héberger la base de connaissance construite autour de notre corpus de documents. L'annotation des documents est un processus comprenant plusieurs tâches dont la plupart peuvent être automatisée. En effet, seul la formalisation du contenu dans un format pivot est délibéremment produite « à la main ». La conversion des fichiers HTML en XHTML est réalisée par un servlet. La transformation du formalisme XML du format ESCRIRE au format TROEPS peut être obtenue par application d'une feuille de style XSL, bien que ceci soit apparement compliqué. On peut aussi envisager une analyse du format ESCRIRE avec *Expat* et produire son équivalent en TROEPS . Cette traduction fait partie des travaux encore non réalisés.

L'interrogation des documents s'effectue au moyen d'une applet permettant de composer des requêtes. La conception de cette interface et son implémentation n'est pas encore effectué. Un évaluateur doit aussi être intégré à TROEPS. Celui-ci permet l'analyse des requêtes transmises en XML et retourne le résultat correspondant toujours en XML.

Conclusion

Le but de notre étude était de représenter la connaissance contenue dans un corpus de documents afin de les indexer par leur contenu. Des requêtes structurées pourront ainsi tirer partie du système formé et rechercher les documents sur des bases sémantiques. Les formalismes de représentation de connaissances constituent donc le point de départ de notre étude. L'action ESCRIRE a d'ailleurs pour objectif d'en comparer trois. Parmi eux, la représentation de connaissances à objets apparaît particulièrement adapté lorsqu'il s'agit de représenter des connaissances complexes sur un domaine en cours d'étude. En effet, les unités de représentation de base sont des objets et sont donc proches de la vision que l'on a d'un domaine. La similarité du méta-langage XML avec les objets a été remarquée et il est donc assez naturel d'utiliser ce langage pour la transcription du formalisme de représentation. Cependant, si XML offre un excellent support syntaxique, il souffre d'une absence de sémantique. Quels sont donc les éléments d'un document à représenter et pour quels types d'applications ?

Nous avons d'abord observé le lien étroit existant entre la nature de la connaissance à représenter et le type du document. En effet, une œuvre de fiction ne sera pas annotée de la même façon qu'un ensemble d'articles scientifiques se référant à un domaine partagé par une communauté. Le choix des éléments à décrire est lui aussi problématique. S'il est clairement impossible de représenter complètement le sens du texte, il faut tout de même fixer le niveau de détail à donner à la représentation. Nous avons ainsi pu voir que plus que le contenu, c'est l'application résultante qui va décider des éléments à représenter. Nous avons donc essayé d'imaginer quels types de requêtes un utilisateur est susceptible de poser, ce qui nous a conduit à proposer un langage de requêtes. Un corpus de travail a été utilisé pour mettre en œuvre les choix effectués. Il concerne les interactions géniques chez la drosophile pendant son processus de segmentation. Le système de représentation de connaissances à objets TROEPS gère les connaissances contenues dans les documents. Un évaluateur de requêtes a été intégré à ce système pour permettre de l'interroger.

Notre travail a abouti à l'identification de certains problèmes intéressants concernant l'intégration d'une représentation formelle de la connaissance contenue dans un document dans le but d'interroger le système résultant. En effet, l'interrogation de ces documents dont on aura préalablement représenté le contenu peut s'effectuer de deux manières différentes. Une première façon, celle qui a été implémentée, consiste à interroger les documents en faisant abstraction du contenu des autres documents. Les inférences ne sont donc que locales au document. Mais on peut considérer que toutes les connaissances doivent être regroupées et former ainsi une base de connaissance. Les requêtes portent alors sur cette base conduisant le système à effectuer des inférences globales au corpus. Cette seconde façon devrait permettre d'augmenter le rappel des réponses retournées. Les solutions proposées n'ont pas toutes été mises en œuvre. Il reste donc à implémenter certaines fonctionnalités afin de tester le système complet sur un corpus important de documents et d'évaluer ses performances par rapport à un moteur de recherche traditionnel plein texte.

Bibliographie

[ACC+97] Augier M., Chouraqui J., Cuny S., Dubost K., Marchandise A., Morelle M. A., and Le Solliec S. Spécification HTML 4.0. EISTI, Juillet 1997. http://www.eisti.fr/eistiweb/docs/normes/html4/cover.html.

[ACY98] Andries P., Cuny S., and Yergeau F. Langage de balisage extensible (XML) 1.0, Février 1998. http://babel.alis.com/web_ml/xml/REC-xml.fr.html.

[BB99] Bosak J. and Bray T. Le langage XML. *Pour la science*, 261, 1999.

[Ber98] Berners-Lee T. Semantic Web Road map. http://www.w3.org/DesignIssues/Semantic.html, 1998.

[BG00] Brickley D. and Guha R. V. Resource Description Framework (RDF) Schema Specification 1.0. W3C, Mars 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.

[Boy99] Boyer F. Une introduction aux servlets. http://sirac.inrialpes.fr/Infos/Personnes/Fabienne.Boyer/cours/GICOM/etape1.html, 1999.

[BPS98] Bray T., Paoli J., and Sperberg-McQueen C. M. Extensible Markup Language (XML) 1.0 Recommandation. W3C, Février 1998. http://www.w3.org/TR/REC-xml.

[CM92] Chein M. and Mugnier M. L. Conceptual graphs : fundamental notions. *Revue d'intelligence artificielle*, 6(4) :365–406, 1992.

[DEFS98] Decker S., Erdmann M., Fensel D., and Studer R. How to Use Ontobroker. The technical core of Ontobroker, 1998.

[DEMN98] Ducournau R., Euzenat J., Masini G., and Napoli A. Langages et modèles à objets : état des recherches et perspectives. Rapport de recherche, INRIA, 1998.

[ESC00] ESCRIRE Action. Méthodologie de comparaison. http://exmo.inrialpes.fr/cooperation/escrire/private/escrire-method.html, 2000.

[Euz99] Euzenat J. Sémantique des représentations de connaissance. Notes de cours de DEA d'Informatique, UJF, Grenoble, 1999.

[Euz00] Euzenat J. Xml est-il le langage de représentation de connaisance de l'an 2000 ? *LMO*, pages 59–74, 2000.

[FAD+98] Fensel D., Angele J., Decker S., Erdmann M., Schnurr H.P., Studer R., and Witt A. On2broker : Lessons Learned from Applying AI to the Web. The new system (long version), 1998.

[FDES98] Fensel D., Decker S., Erdmann M., and Studer R. How to make the WWW Intelligent. In *The 11th Banff Knowledge Acquisition for Knowledge - Based System Workshop (KAW98)*, Banff, Canada, Avril 1998.

[FES98] Fensel D., Erdmann M., and Studer R. Ontobroker : The Very High Idea. In *The 11th International Flairs Conference (FLAIRS-98)*, Sanibal Island, Floride, Mai 1998.

[FMR00] Fankhauser P., Marchiori M., and Robie J. XML Query Requirements, Janvier 2000. http://www.w3.org/TR/xmlquery-req.

[Gru93] Gruber T. R. A translation approach to portable ontology specifications. In *Knowledge Acquisition*, pages 199–220, 1993.

[Heb99] Hebert C. Modèle de traitement de RDF basé sur les Graphes Conceptuels. Master's thesis, INRIA Sophia Antipolis, Juin 1999.

[HFB+00] Horrocks I., Fensel D., Broekstra J., Decker S., Erdmann M., Goble C., Van Harmelen F., Klein M., Staab S., and Studer R. The Ontology Interchange Language OIL. http://www.ontoknowledge.com/oil, 2000.

[HHL99] Heflin J., Hendler J., and Luke S. A Knowledge Representation Language for Internet Applications. Technical report, Dept. of Computer Science, University of Maryland at College Park, 1999.

[Kay97] Kayser D. *La représentation des connaissances.* Hermès, 1997.

[KCT99] Karp P. D., Chaudhri V. K., and Thomere J. XOL : An XML-Based Ontology Exchange Language. http://www.ai.sri.com/pubs/technotes/aic-tn-1999 :559/, 1999.

[Ken] Kent R. Ontology Markup Language - Conceptual Knowledge Markup Language. http://www.ontologos.org/.

[Laz98] Lazinier E. XML expliqué aux débutants. http://www.chez.com/xml/initiation/index.htm, 1998.

[LS99] Lassila O. and Swick R. R. Resource Description Framework (RDF) Model and Syntax Specification. W3C, Février 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[LSRH97] Luke S., Spector L., Roger D., and Hendler J. Ontology-based Web Agents. In ACM Association for Computing Machinery, editor, *The 1st International Conference on Autonomous Agents (AA-97)*, pages 59–66, 1997.

[Mai98] Maier D. Database Desiderata for an XML Query Language. In *The Query Languages Workshop (QL'98)*, 1998.

[MC96] Mugnier M. L. and Chein M. Représenter des connaissances et raisonner avec des graphes. *Revue d'intelligence artificielle*, 10(1) :7–56, 1996.

[Min75] Minsky M. A framework for representing knowledge. In Winston P., editor, *The psychology of computer vision.* McGraw-Hill, 1975.

[MMA98] Mecca G., Merialdo P., and Atzeni P. Do we really need a new query language for XML ? In *The Query Languages Workshop (QL'98)*, 1998.

[Nap97] Napoli A. Une introduction aux logiques de description. Rapport de Recherche 3314, INRIA Lorraine, Nancy (FR), 1997.
ftp://ftp.inria.fr//INRIA/publication/publi-ps-gz/RR/RR-3314.ps.gz.

[Neb90] Nebel B. Reasoning and revision in hybrid representation systems. In *Lecture Notes in Artificial Intelligence*, volume 422. Springer Verlag, Berlin (DE), 1990.

[NED00] Napoli A., Euzenat J., and Ducournau R. Les représentations des connaissances par objets. *Technique et science informatiques*, 19(1), 2000.

[ODM] ODMG (Object Data Managment Group). A Standard for Object Storage.
http://www.odmg.org.

[Qui68] Quillian J. R. Semantic memory. In *Semantic Information Processing*, pages 227–270. MIT Press, Cambridge, MA, 1968.

[RLJ00] Raggett D., Le Hors A., and Jacobs I. XHTML 1.0 : The Extensible HyperText Markup Language. W3C, Janvier 2000.
http://www.w3.org/TR/xhtml1/.

[SHE95] SHERPA Project. Troeps 1.0 reference manual. INRIA Rhône-Alpes, Grenoble (FR), 1995.

[Sow84] Sowa J. *Conceptual structures : information processing in mind and machine.* Addisson-Wesley, Reading (MA US), 1984.

[Val99] Valtchev P. *Construction automatique de taxonomies pour l'aide à la représentation de connaissances par objets.* PhD thesis, Université Joseph Fourier - Grenoble 1, 1999.

[VF99] Van Harmelen F. and Fensel D. Practical Knowledge Representation for the Web. In *IJCAI'99 : Workshop on Intelligent Information Integration*, 1999.

[Wal97] Walsh N. A Guide to XML. http://www.xml.com/pub/w3j/s3.walsh.html, Octobre 1997.

Annexe A

Documents issus de Medline

A.1 Un texte issu de notre corpus de documents

Control of the initiation of homeotic gene expression by the gap genes giant and tailless in Drosophila.

Reinitz J, Levine M

Department of Biological Sciences, Fairchild Center, Columbia University, New York, New York 10027.

The process of segmentation in Drosophila is controlled by both maternal and zygotic genes. Members of the gap class of segmentation genes play a key role in this process by interpreting maternal information and controlling the expression of pair-rule and homeotic genes. We have analyzed the pattern of expression of a variety of homeotic, pair-rule, and gap genes in tailless and giant gap mutants. tailless acts in two domains, one anterodorsal and one posterior. In its anterior domain tailless exerts a repressive effect on the expression of fushi tarazu, hunchback, and Deformed. In its posterior domain of action, tailless is responsible for the establishment of Abdominal-B expression and demarcating the posterior boundary of the initial domain of expression of Ultrabithorax. giant is an early zygotic regulator of the gap gene hunchback: in giant- embryos, alterations in the anterior domain of hunchback expression are visible by the beginning of cycle 14. giant also regulates the establishment of the expression patterns of Antennapedia and Abdominal-B. In particular, giant is the factor that controls the anterior limit of early Antennapedia expression.

PMID: 1972684, UI: 90292349

A.2 Informations complémentaires fournies par Medline pour ce même document

1 : Reinitz J, et al Control of...[PMID:1972684]

```
UI  - 90292349
AU  - Reinitz J
AU  - Levine M
TI  - Control of the initiation of homeotic gene expression by the gap genes
      giant and tailless in Drosophila.
LA  - Eng
MH  - Animal
MH  - Cleavage Stage, Ovum/physiology
MH  - Drosophila/embryology/*genetics
MH  - DNA Probes
MH  - *Gene Expression Regulation
MH  - *Genes, Homeobox
MH  - Genes, Regulator/*physiology
MH  - Microscopy
MH  - Transcription, Genetic
RN  - 0 (DNA Probes)
PT  - JOURNAL ARTICLE
DP  - 1990 Jul
IS  - 0012-1606
TA  - Dev Biol
PG  - 57-72
CY  - UNITED STATES
IP  - 1
VI  - 140
JC  - E7T
AA  - Author
EM  - 199010
AD  - Department of Biological Sciences, Fairchild Center, Columbia University,
      New York, New York 10027.
PMID- 0001972684
CU  - 1994
MHDA- 1990/07/01 00:00
SO  - Dev Biol 1990 Jul;140(1):57-72
```

Annexe B

DTDs utilisées

B.1 DTD définissant le format pivot ESCRIRE

```
<?xml version="1.0" encoding="UTF-8" ?>
<!- escrire.dtd version="0.4" last-modification="13/03/2000"
           url="http://escrire.inrialpes.fr/escrire/escrire.dtd" ->
<!- ========================================================================== ->
<!ELEMENT esc:ontology ((esc:descclass | esc:defclass
                        |esc:descrelation | esc:defrelation
                        |esc:descbinrel | esc:defbinrel
                        |esc:object | esc:relation)*)>
<!ATTLIST esc:ontology url CDATA #REQUIRED
                       xmlns:esc CDATA #FIXED "http://escrire.inrialpes.fr/">
<!ELEMENT esc:descclass (esc:classref?, esc:defattribute*)>
<!ATTLIST esc:descclass name CDATA #REQUIRED>
<!ELEMENT esc:defclass (esc:classref?, esc:defattribute*)>
<!ATTLIST esc:defclass name CDATA #REQUIRED>
<!ELEMENT esc:classref EMPTY>
<!ATTLIST esc:classref name CDATA #REQUIRED>
<!ELEMENT esc:descrelation (esc:relref?, (esc:defattribute | esc:defrole)*)>
<!ATTLIST esc:descrelation name CDATA #REQUIRED>
<!ELEMENT esc:defrelation (esc:relref?, (esc:defattribute | esc:defrole)*)>
<!ATTLIST esc:defrelation name CDATA #REQUIRED>
<!ELEMENT esc:descbinrel (esc:relref?, (esc:defattribute | esc:defrole)*)>
<!ATTLIST esc:descbinrel name CDATA #REQUIRED
                         converse CDATA #IMPLIED
                         symmetric (yes | no) "no"
                         transitive (yes | no) "no"
                         reflective (yes | no) "no"
                         antisymmetric (yes|no) "no">
<!ELEMENT esc:defbinrel (esc:relref?, (esc:defattribute, esc:defrole)*)>
<!ATTLIST esc:defbinrel name CDATA #REQUIRED
                        converse CDATA #IMPLIED
                        symmetric (yes | no) "no"
                        transitive (yes | no) "no"
                        reflective (yes | no) "no"
                        antisymmetric (yes | no) "no">
<!ELEMENT esc:relref EMPTY>
<!ATTLIST esc:relref name CDATA #REQUIRED>
<!ELEMENT esc:defattribute ((esc:classref | esc:typeref)*)>
<!ATTLIST esc:defattribute name CDATA #REQUIRED
                           cons (set | list | bag | none) "none">
<!ELEMENT esc:defrole (esc:classref*)>
```

```
<!ATTLIST esc:defrole name CDATA #REQUIRED>
<!ELEMENT esc:typeref EMPTY>
<!ATTLIST esc:typeref name CDATA #REQUIRED>
<!- ========================================================================== ->
<!ELEMENT esc:content ((esc:object | esc:relation | esc:objref)*)>
<!ATTLIST esc:content url CDATA #REQUIRED
                      xmlns:esc CDATA #FIXED "http://escrire.inrialpes.fr/"
                      ontology CDATA #REQUIRED>
<!ELEMENT esc:object (esc:attribute*)>
<!ATTLIST esc:object type CDATA #REQUIRED
                     id CDATA #REQUIRED>
<!ELEMENT esc:objref EMPTY>
<!ATTLIST esc:objref id CDATA #REQUIRED>
<!ELEMENT esc:relation ((esc:attribute | esc:role)*)>
<!ATTLIST esc:relation type CDATA #REQUIRED>
<!ELEMENT esc:role (esc:objref+)>
<!ATTLIST esc:role name CDATA #REQUIRED>
<!ELEMENT esc:attribute ANY>
<!ATTLIST esc:attribute name CDATA #REQUIRED>
```

B.2 DTD définissant le langage de requête

```
<?xml version="1.0" encoding="UTF-8" ?>
<!- query.dtd version="0.4" last-modification="09/05/2000"
            url="http://escrire.inrialpes.fr/dtd/query.dtd" ->
<!- import escrire DTD ->
<!ENTITY % escrireDTD SYSTEM "http://escrire.inrialpes.fr/dtd/escrire.dtd">
%escrireDTD;
<!- declaration des entites ->
<!ENTITY esc:expr "esc:not | esc:and | esc:or | esc:exist | esc:all | esc:eq">
<!ELEMENT esc:query (esc:select, esc:from, esc:where?, esc:orderby?)>
<!ATTLIST esc:query url CDATA #REQUIRED
                    xmlns:esc CDATA #FIXED "http://escrire.inrialpes.fr/">
<!ELEMENT esc:select (esc:path*)>
<!ELEMENT esc:from ((esc:objref | esc:objvar | esc:relvar)+)>
<!ELEMENT esc:where (&esc:expr;)>
<!ELEMENT esc:orderby (esc:path+)>
<!ELEMENT esc:objvar EMPTY>
<!ATTLIST esc:objvar type CDATA #REQUIRED
                     id CDATA #REQUIRED>
<!ELEMENT esc:relvar EMPTY>
<!ATTLIST esc:relvar type CDATA #REQUIRED
                     id CDATA #REQUIRED>
<!ELEMENT esc:var EMPTY>
<!ATTLIST esc:var name CDATA #REQUIRED>
<!ELEMENT esc:not (&esc:expr;)>
<!ELEMENT esc:and ((&esc:expr;)+)>
<!ELEMENT esc:or ((&esc:expr;)+)>
<!ELEMENT esc:exist (esc:var, &esc:expr;)>
<!ELEMENT esc:all (esc:var, &esc:expr;)>
<!ELEMENT esc:eq (esc:path, (esc:value | esc:objvarref | objref))>
<!ELEMENT esc:path ((esc:objvarref | esc:relvarref),esc:attribute)>
<!ELEMENT esc:objvarref EMPTY>
<!ATTLIST esc:objvarref type CDATA #REQUIRED
                        id CDATA #REQUIRED>
<!ELEMENT esc:relvarref EMPTY>
<!ATTLIST esc:relvarref type CDATA #REQUIRED
                        id CDATA #REQUIRED>
```

```
<!ELEMENT esc:attribute EMPTY>
<!ATTLIST esc:attribute name CDATA #REQUIRED>
<!ELEMENT esc:value ANY>
```

B.3 DTD définissant le format des réponses

```
<?xml version="1.0" encoding="UTF-8" ?>
<!- resc.dtd version="0.2" last-modification="09/05/2000"
           url="http://escrire.inrialpes.fr/dtd/resc.dtd" ->
<!ENTITY % esc SYSTEM "http://escrire.inrialpes.fr/dtd/escrire.dtd">
%esc;
<!ENTITY % qesc SYSTEM "http://escrire.inrialpes.fr/dtd/qesc.dtd">
%qesc;
<!ELEMENT esc:resc (esc:query, (esc:answer* | esc:noanswer))>
<!ATTLIST esc:resc ontology CDATA #REQUIRED
                   xmlns:esc CDATA #FIXED "http://escrire.inrialpes.fr/">
<!ELEMENT esc:answer (esc:vref*, esc:docref+)>
<!ELEMENT esc:noanswer EMPTY>
<!ELEMENT esc:docref EMPTY>
<!ATTLIST esc:docref href CDATA #REQUIRED>
<!ELEMENT esc:vref #PCDATA>
<!ATTLIST esc:vref name CDATA #REQUIRED>
```

Annexe C

Une représentation exhaustive du résumé fourni en Annexe A.1

```
<?XML version="1.0"?>
<formalisation>
   <relation action="control">
      <sujet>
         <AND>
            <entite nom="maternal genes" />
            <entite nom="zygotic genes" />
         </AND>
      </sujet>
      <objet>
         <processus nom="segmentation" />
         <IN>
            <entite nom="Drosophila" />
         </IN>
      </objet>
   </relation>
   <role action="play" qualification="key">
      <sujet>
         <entite nom="members" />
         <OF>
            <entite nom="gap class" />
         </OF>
      </sujet>
      <objet>
         <processus nom="segmentation" />
      </objet>
   </role>
   <relation action="interpret">
      <sujet>
         <entite nom="members" />
         <OF>
            <entite nom="gap class" />
         </OF>
      </sujet>
      <objet>
         <entite nom="maternal information" />
      </objet>
   </relation>
   <relation action="control">
      <sujet>
```

```
      <entite nom="members" />
      <OF>
         <entite nom="gap class" />
      </OF>
   </sujet>
   <objet>
      <entite nom="expression" />
      <OF>
         <AND>
            <entite nom="pair-rule genes" />
            <entite nom="homeotic genes" />
         </AND>
      </OF>
   </objet>
</relation>
<relation action="analyse">
   <sujet>
      <entite nom="we" />
   </sujet>
   <objet>
      <entite nom="pattern expression" />
      <OF>
         <entite nom="variety" />
         <OF>
            <AND>
               <entite nom="homeotic genes" />
               <entite nom="pair-rule genes" />
               <entite nom="homeotic genes" />
            </AND>
         </OF>
      </OF>
      <IN>
         <AND>
            <entite nom="giant gap mutant" />
            <entite nom="tailless gap mutant" />
         </AND>
      </IN>
   </objet>
</relation>
<relation action="act">
   <sujet>
      <entite nom="tailless" />
   </sujet>
   <objet>
      <IN>
         <AND>
            <entite nom="anterodorsal domain" />
            <entite nom="posterior domain" />
         </AND>
      </IN>
   </objet>
</relation>
<relation action="exert">
   <sujet>
      <entite nom="tailless" />
   </sujet>
   <objet>
      <effet qualification="repressive" />
      <ON>
```

```
         <entite nom="expression" />
         <OF>
            <AND>
               <entite nom="fushi tarazu" />
               <entite nom="hunchback" />
               <entite nom="Deformed" />
            </AND>
         </OF>
      </ON>
      <IN>
         <entite nom="anterior domain" />
      </IN>
   </objet>
</relation>
<relation action="responsible for">
   <sujet>
      <entite nom="tailless" />
   </sujet>
   <objet>
      <AND>
         <entite nom="establishment of expression" />
         <OF>
            <entite nom="Abdominal-B" />
         </OF>
         <role action="demarcating" />
            <objet>
               <entite nom="boundary" />
               <OF>
                  <entite nom="initial domain of expression" />
                  <OF>
                     <entite nom="Ultrabithorax" />
                  </OF>
               </OF>
            </objet>
         </role>                  </AND>
      <IN>
         <entite nom="posterior domain" />
      </IN>
   </objet>
</relation>
<relation action="is">
   <sujet>
      <entite nom="giant" />
   <sujet>
   <objet>
      <entite nom="zygotic regulator" qualification="early" />
      <OF>
         <entite nom="gap gene hunchback" />
      </OF>
   </objet>
</relation>
<relation action="are visible">
   <sujet>
      <entite nom="alterations" />
      <OF>
         <entite nom="hunchback expression" />
      </OF>
      <IN>
         <entite nom="anterior domain" />
```

```
            </IN>
         </sujet>
         <objet>
            <IN>
               <entite nom="giant embryos" />
            </IN>
            <BY>
               <entite nom="beginning of cycle 14" />
            </BY>
         </objet>
      </relation>
      <relation action="regulate">
         <sujet>
            <entite nom="giant" />
         </sujet>
         <objet>
            <entite nom="establishment of expression patterns" />
            <OF>
               <AND>
                  <entite nom="Antennapedia" />
                  <entite nom="Abdominal-B" />
               </AND>
            </OF>
         </objet>
      </relation>
      <relation action="control">
         <sujet>
            <entite nom="giant" />
         </sujet>
         <objet>
            <entite nom="anterior limit" />
            <OF>
               <entite nom="Antennapedia expression" qualification="early" />
            </OF>
         </objet>
      </relation>
   </formalisation>
```

Annexe D

Serveur de servlets

Un *servlet* est un module pouvant s'exécuter à l'intérieur d'un serveur Web. Les servlets sont un substitut aux scripts CGI en étant plus simples à développer et plus rapides à l'exécution. En effet, le mécanisme CGI nécessite pour chaque requête le démarrage d'un processus « lourd ». Les servlets s'exécutent comme des *threads* dans un même interprête. Les changements de contexte sont donc plus rapides. En fait, les servlets sont aux serveurs ce que les *applets* sont aux navigateurs. Un servlet peut être chargé automatiquement lors du démarrage du serveur Web ou lors de la première requête du client. Mais une fois chargés, les servlets restent actifs dans l'attente d'autres requêtes du client.

Grâce à la création d'un environnement de prestation de services requête/réponse via le Web, les servlets permettent l'extension des fonctions du serveur. Lorsqu'un client envoie une requête au serveur, ce dernier transmet au servlet les informations relatives à la requête. Par la suite, le servlet crée une réponse que le serveur renvoie au client. Lors de la création de la réponse, le servlet peut utiliser toutes les fonctions du langage JAVA ou communiquer avec des ressources externes (fichier, bases de données ou applications écrites en JAVA ou dans d'autres langages).

Le principe d'appel d'un servlet est le suivant. Par défaut, étant donné le nom d'une requête `N`, le serveur Web recherche une classe de nom `N` et l'exécute. Par exemple, la requête "http ://escrire.inrialpes.fr/servlets/Hello" entraine l'exécution du servlet `Hello`, c'est-à-dire de la classe `Hello.java` qui est située dans le répertoire `servlets` du serveur Web. La méthode appelée dépend du type de la requête HTTP (GET, POST, PUT, etc). Bien entendu, un serveur Web doit tourner sur la machine `escrire` et être connecté au port `80` (port par défaut).

Traitement de requêtes HTTP

Le diagramme de classes simplifié de l'interface *Servlet* est donné dans la Figure D.1.

Lorsque le serveur charge un servlet pour la première fois, il exécute la méthode *init*, qui est implémentée par la classe abstraite *GenericServlet* et qui peut être surchargée par l'application pour effectuer certains traitements. La méthode *service* est implémentée par la classe *HttpServlet*. La classe *HttpServlet* doit être étendue pour développer des servlets répondant à des requêtes HTTP. Le protocole HTTP définit un mode simple de communication selon le modèle requête-réponse. Une requête HTTP est principalement composée du nom d'une méthode (action à réaliser) et d'une URL désignant l'objet recherché (document html, script à exécuter...). Les méthodes disponibles sont présentées dans le tableau D.1.

Pour l'exemple du servlet `Hello`, seule la méthode *doGet* est utilisée. L'accès à l'URL "http ://escrire.inrialpes.fr/servlets/Hello" engendrera automatiquement l'appel de la méthode *doGet*

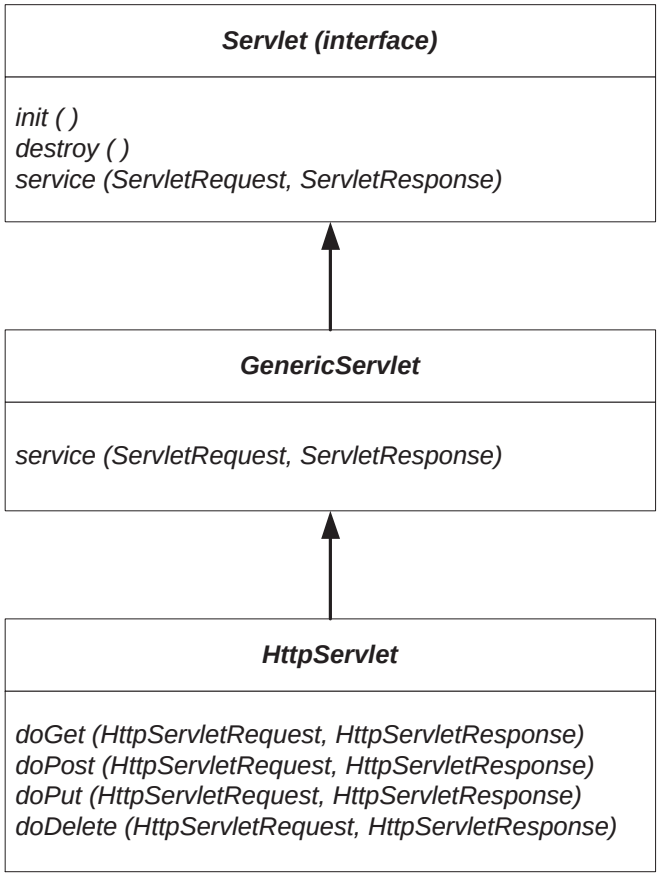

FIG. D.1 – *Diagramme de classes simplifié de l'interface* Servlet

Méthodes	Actions
GET	*Retourne l'objet*[a] *spécifié. Dans le cas où l'objet est un servlet, la méthode doGet est exécutée*
POST	*Demande de stockage d'une information. doPost est exécutée. Les paramètres d'appel sont passés dans le flux d'entrée (InputStream)*
PUT	*Envoie une nouvelle copie d'un objet existant au serveur (beaucoup de serveurs refusent cette méthode)*
DELETE	*Détruit l'objet de manière irréversible (beaucoup de serveurs refusent cette méthode)*

TAB. D.1 – *Les méthodes principales de la classe* HttpServlet

[a] le terme *objet* fait référence à un document au sens large (html, script CGI, bytecode java ...)

sur le servlet `Hello`. Si cet servlet est chargé pour la première fois dans le serveur Web, alors la méthode *init* sera appelée avant l'appel de *doGet*. Depuis un formulaire, il en sera de même :

```
<form method="post" action="http://escrire.inrialpes.fr/servlets/Hello">
...
<input type="text" name="URL1" size="20" />
...
</form>
```

engendre l'appel automatique de la méthode *doGet* du servlet `Hello`, lorsque le formulaire est validé par le client.

Récupération de paramètres

L'interface *ServletRequest* permet à un servlet d'accéder aux paramètres passés par le client, au protocole utilisé et aux noms des machines cliente et serveur. La sous-classe *HttpServletRequest* permet d'accéder à des informations spécifiques au protocole HTTP comme par exemple les paramètres d'appel ou les cookies présents dans l'en-tête des requêtes. Par exemple :

`Coté client = envoi de la requête`
"http://escrire.inrialpes.fr/servlets/TransformServlet?stylesheet=http://escrire.inrialpes.fr/xsl/xhtml2esc.xsl"
`Coté serveur`

```
public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {
    ...
    for (Enumeration e = req.getParameterNames() ; e.hasMoreElements() ;) {
        /* Récupération des paramètres */
        String attribute = (String)e.nextElement() ;
        ...
    }
}
```

Retour de résultats

L'interface *HttpServletResponse* fournit à un servlet des méthodes permettant de fixer le contenu MIME du contenu (`text/html`, `image/gif`, `application/postcript` ...). Ce contenu est envoyé au client, par exemple, au travers d'un flux *ServletOutputStream* :

```
public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {
    ...
    ServletOutputStream out = res.getOutputStream() ;
    res.setContentType("text/html") ;
    ...
}
```

Résumé

Actuellement, le Web contient d'importantes quantités d'informations couvrant tous les sujets imaginables. Le problème qui était avant de savoir si une information, même très spécifique, était disponible sur le Web, est maintenant devenu comment retrouver cette information. Apporter du sens intelligible et exploitable par des machines aux documents devrait leur permettre d'utiliser l'information présente, d'amĺiorer les techniques de recherche, et donc de faire du Web une gigantesque base de connaissance. Les langages de représentation de connaissance sont de bons candidats si l'on souhaite décrire le contenu de documents. L'action ESCRIRE a d'ailleurs pour objectif d'en comparer trois. Parmi eux, la représentation de connaissances à objets apparaît particulièrement adapté lorsqu'il s'agit de représenter des connaissances complexes sur un domaine en cours d'étude. On pourra alors manipuler plus efficacement une base de documents en les indexant par leur contenu (ou leur sens). Les documents pertinents seront ramenés à partir de requêtes structurées tirant parti du formalisme de représentation de connaissance (hiérarchie de classes, mécanismes de classification...).

Nous avons d'abord observé le lien étroit existant entre la nature de la connaissance à représenter et le type du document. Nous avons aussi pu voir que plus que le contenu, c'est l'application résultante qui va décider des éléments à représenter. Nous avons donc essayé d'imaginer quels types de requêtes un utilisateur est susceptible de poser, ce qui nous a conduit à proposer un langage de requêtes. Un corpus de travail a été utilisé pour mettre en œuvre les choix effectués. Il concerne les interactions géniques chez la drosophile pendant son processus de segmentation. Le système de représentation de connaissances à objets TROEPS gère les connaissances contenues dans les documents. Un évaluateur de requêtes a été intégré à ce système pour permettre de l'interroger.

Mots-clés: Représentation de connaissance, Représentation du contenu, Sémantique, XML, Recherche d'information, Annotations, Ontologies, Objets.