La reécriture et l'évaluation de requêtes arbres avec XPath

Rewriting and Evaluating Tree Queries with XPath

B. AMANN[1], C. BEERI[2], I. FUNDULAKI[1], M. SCHOLL[1] et A-M. VERCOUSTRE[3]

[1] CEDRIC-CNAM, 2 rue Conté, 75003 Paris, France
[2] The Hebrew University of Jerusalem, 91904 Jerusalem, Israel
[3] CSIRO Mathematical and Information Sciences, Carlton 723 Swanson Street, Carlton, VIC 3053, Australia

E-mail: {amann,fundulak,scholl}@cnam.fr, beeri@cs.huji.ac.il, Anne-Marie.Vercoustre@cmis.csiro.au

Résumé

C-Web est un ensemble d'outils pour une communauté d' utilisateurs qui partagent des informations associées à un domaine spécifique à travers le Web. Le composant essentiel de ce systéme est une ontologie qui est partagée entre les membres de la communauté et utilisée comme interface commun pour l'intégration, interrogation et échange des informations. Dans cet article nous décrivons un langage simple mais puissant pour l'intégration de documents XML dans C-Web. Plus précisément, nous illustrons comment on peut exploiter et intégrer de serveurs Web capable de répondre à des requêtes XPath. Cet intégration est faite par des règles d'association de chemins d'éxpression XPath aux concepts et roles de l'ontologie. Nous décrivons un algorithme de reécriture de requêtes qui utilise ces règles pour reformuler une requête utilisateur à un ensemble de requêtes XPath. En outre, on décrit des techniques d'évaluation qui nous permettent de réduire le volume des fragments XML envoyés par les sources pendant l'évaluation d'une requête.

Mots clés : *communauté web, ontologies, XML, XPath,règles d'association, traduction de requêtes*

Abstract

A Community Web portal is a set of tools for a community of people who want to share information on a certain domain via the Web. The backbone of this system is an ontology which is understood by all members of the community and used as the common interface component for integrating, querying, structuring and exchanging information. This paper describes a simple but nevertheless powerful language for integrating XML documents into a Community Web system. More precisely, we describe how to add and exploit XPath enabled Web servers by mapping standard XPath location paths to conceptual paths in the system ontology. We present a query rewriting algorithm using such mappings. It transforms a user query into a set of XML queries based on XPath patterns for selecting XML fragments. Finally, we describe some evaluation techniques for reducing the size of XML data returned by the XML source for query evaluation.

Keywords: *community web, ontology, XML, XPath, mapping rules, query translation*

1 Introduction

The C-Web (Community Web) project [1] [3, 2] aims at supporting the sharing, integration and retrieval of information in a specific domain of interest. The main objective is to provide to a group of people who desire to access and exchange knowledge and information in this domain, the infrastructure for *publishing* information sources and formulating *structured queries* by taking into consideration the conceptual representation of the domain in form of an *ontology*.
In our first C-Web prototype [3] we have proposed a *(content) description language* to describe (index) the contents of Web resources (identified by URLs) in terms of a *domain specific ontology extended with specialized thesauri* [4]. Resource descriptions are stored in a *description base* and the result to some user query is a list of URLs. The actual source contents and structure are completely ignored during query evaluation. This has the advantage that it is possible to semantically index any kind of Web resources (including images) which are not necessarily structured XML documents or database records. But an obvious shortcoming of this approach is that manually indexing a large collection of structured documents (e.g. XML documents in an XML repository) might be a difficult and tedious task, whereas similar, and often better results, could be obtained by issuing structured queries (e.g. XML queries) against the XML repository.
In this paper we are interested in the querying of XML resources. More precisely, we want to take advantage of the structure of XML resources, generally described by a Document Type Definition (DTD) or an XML schema, for mapping XML fragments to concepts and roles in the ontology. There are two orthogonal objectives here: 1) to be able to translate and forward semantic user queries to diverse XML repositories while hiding their heterogeneity, and 2) to limit the materialization of descriptions on the C-Web repository when the information is available on the source. The second point is particularly relevant when the values of some elements change over time.
In the following, we will present :

- a *mapping language* relating XML fragments described by *XPath location paths* to the concepts and roles of an ontology,

[1] http://cweb.inria.fr

- a simple *tree query language* for C-Web portals,
- a *query rewriting algorithm* for translating tree queries into XML queries and
- a *query evaluation strategy* for XPath enabled XML resources.

The paper is organized as follows. In Section 2 we present a cultural application example to illustrate our approach. We informally describe the mapping rules that can be defined between XML resources and a cultural ontology and we illustrate how these rules can be exploited for translating and evaluating user queries. Related work is discussed in Section 3. In Section 4 we present the mapping language for XML resources and the semantics of mapping rules. Tree queries and the rewriting algorithm for translating tree queries according to a given set of mapping rules are presented in Section 5. In Section 6 we describe a query evaluation strategy for XPath enabled XML resources. Finally, Section 7 contains a conclusion and future work.

2 System Overview through a Cultural Example

We illustrate our approach considering the integration of cultural information sources accessible from Web servers. Suppose that *http://www.art.com* is a web server of XML documents about artists and art in general. An example of an XML document we would like to query and the corresponding DTD are shown in Figures 1 and 2.

```
<ARTIST SCHOOL='DUTCH'>
  <NAME>Vincent Van Gogh</NAME>
  <NATIONALITY>Dutch</NATIONALITY>
  <ARTIFACT FORM='P'>
    <TITLE>Church at Auvers</TITLE>
    <MATERIAL>Oil on Canvas</MATERIAL>
    <LOCATION>
      Musee d'Orsay, Paris France
    </LOCATION>
  </ARTIFACT>
  <ARTIFACT FORM='P'>
    <TITLE>Vase With Flowers</TITLE>
    <MATERIAL>Oil on Canvas</MATERIAL>
    <LOCATION>
      Van Gogh Museum, Amsterdam
    </LOCATION>
  </ARTIFACT>
</ARTIST>
```

Figure 1: XML document about Vincent Van Gogh

An example of an ontology for cultural artifacts, which is inspired from the ICOM/CIDOC Reference Model[2], is shown in Fig. 3 as a labeled graph. Nodes of the graph correspond to the *concepts* of the ontology which are connected by *binary roles* and simple *inheritance* (isa) links.

[2] http://cidoc.ics.forth.gr/crm_intro.html

Ten concepts and ten roles describe actors (persons) performing activities for producing artifacts. Roles are represented by solid arcs and each role has an inverse role which is defined within parentheses. Dashed arcs represent concept inheritance. For example, concept Person is a subconcept of Actor and inherits role *performed* (inverse role *performed_by*) that relates actors (instances of concept Actor) to activities (instances of concept Activity). Observe that we do not distinguish concept attributes from roles. For example, role *has_name* defined on concept Person returns the name of a person in form of an instance of concept (type) String.

```
 1. <!ELEMENT ARTIST
        (NAME,NATIONALITY,ARTIFACT*)>
 2. <!ELEMENT NATIONALITY  (#PCDATA)>
 3. <!ATTLIST ARTIST SCHOOL
 4.               (ITALIAN|DUTCH|OTHER)>
 5. <!ELEMENT NAME  (#PCDATA)>
 6. <!ELEMENT ARTIFACT
           (TITLE,MATERIAL,LOCATION)>
 7. <!ATTLIST ARTIFACT FORM (P|S|G)>
 8. <!ELEMENT TITLE (#PCDATA)>
 9. <!ELEMENT LOCATION (#PCDATA)>
10. <!ELEMENT MATERIAL (#PCDATA)>
```

Figure 2: An XML DTD for Artists

2.1 Mapping XML Resources

Typically source descriptions as described in [3] are useful for storing information which cannot be extracted either from the document contents or its structure. Then the description base might help the user to discover sources of information relevant to her query. But if we want to take advantage of the document contents in a collection of XML documents, a user query should be translated into an XML query. This can be done if there exists a mapping between the document structure on the one hand and concepts and roles in the ontology on the other hand.

As mentioned in [10] there exist different ways for defining such mappings depending on the size and preciseness of the mapping definition but also the complexity of the query rewriting algorithm. Among the different possibilities (node-to-node, path-to-path, tree-to-tree, etc.), we have chosen the path-to-path approach and introduce rules that map *XPath location paths* to *conceptual paths* in the conceptual schema. The choice of XPath as part of our mapping language is justified in more detail in Section 3.

For example, the rules illustrated in Fig. 4 map XML resources as described by the DTD of Fig. 2 to paths in the ontology of Fig. 3. The left hand side of a mapping rule is called the *source path* of the rule and considers an XPath *location path* [8] (see Section 4.1 for more details about XPath) evaluated on the context defined by a concrete URL or a variable. The right hand side of a mapping rule is a path in the conceptual schema called the *schema path* of the rule.

Informally, given such rules, the interpretation of a concept

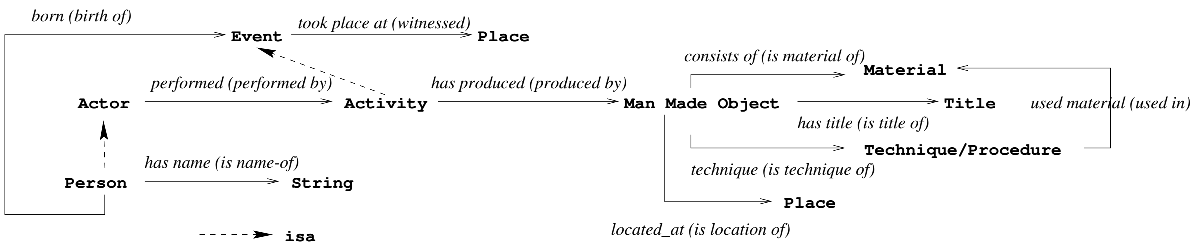

Figure 3: An Ontology for Cultural Artifacts

R_1:	`http://www.art.com//ARTIST` as u_1	$\leftarrow$	Person
R_2:	u_1`/NAME`	$\leftarrow$	has_name
R_3:	u_1`/ARTIFACT` as u_2	$\leftarrow$	performed.has_produced
R_4:	u_2`/TITLE`	$\leftarrow$	has_title
R_5:	u_2`/PROCEDURE` as u_3	$\leftarrow$	technique
R_6:	u_3`/MATERIAL`	$\leftarrow$	used_material
R_7:	u_1`/NATIONALITY`	$\leftarrow$	born.took_place_at
R_8:	`http://www.art.com//ARTIFACT` as u_2	$\leftarrow$	Man Made Object
R_9:	u_2`/LOCATION`	$\leftarrow$	located_at

Figure 4: Set of Mapping Rules

in the ontology is a set of XML fragments. More formally, a rule $r : \ a/q \ as \ b \leftarrow p$ where a is a URL or a variable, b is a variable, q is an XPath location path and p is a schema path, is interpreted as *"the XML fragments obtained by XPath q starting from URL or variable a are bound to variable b and belong to the set of instances (population) of the concept reached by schema path p"*. For example, the first mapping rule states that all elements of type `ARTIST` and descendants of the root elements of the XML documents in *http://www.art.com* are bound to variable u_1 and are instances of concept Person.

In the same way, rule R_8 states that all elements of type `ARTIFACT` and descendants of the root elements of XML documents in *http://www.art.com* are instances of concept Man Made Object.

Rules also define role instances. For example, rule R_2 specifies that all sub-elements obtained by evaluating XPath */NAME* on some fragment x obtained by rule R_1 correspond to a string (instance of concept String) which is the name of person x (fragments obtained by rule R_1 are defined as instances of concept Person). Rule R_3 creates instances of role $performed.has\_produced$ connecting each element (person) x obtained by rule R_1 to all fragments (artifacts) that can be obtained from x by applying XPath */ARTIFACT*.

Observe that, in order to be consistent with the conceptual schema, the concatenation of the schema paths of rules R_1 and R_2 must be a path in the conceptual schema and by concatenating source paths of rules R_1 and R_2, we obtain a new rule $R_1.R_2$: `http://www.art.com//ARTIST/NAME` $\leftarrow$Person.has_name. In general, a set of k_1 rules that bind the variable u, and a set of k_2 rules that use it can be extended by a set of $k_1 * k_2$ rules. That is, the resulting set of mapping rules after expansion may be exponentially larger than the set of rules with variables.

The use of variables is a powerful tool that helps to reduce the size of mappings, sometimes quite drastically, while making the structure easier to understand. The use of different variables enables separating contexts in which different mapping rules can apply, or to map different XPaths to the same variable. To illustrate the latter, consider rules R_8 and R_3 which have been both assigned to variable u_2. By this multiple assignment, rules R_4 and R_5 can be applied to all fragments found by using rule R_3 and rule R_8. On the other hand, if R_8 had been assigned a new variable, say $u_4 \neq u_2$, rules R_4 and R_5 could not be used for finding the title and techniques of man-made objects found by rule R_8.

2.2 Query Rewriting and Evaluation

The basic motivation behind (i) describing sources or/and (2) producing mappings is to provide users with more powerful query facilities. Queries are formulated in an OQL-like syntax (Section 5.1) considering the ontology as a standard object-oriented database schema.

The following example illustrates the evaluation of a user query. Suppose the user asks the titles of artifacts created by the artist Picasso. The user query is illustrated in Fig. 5.

select	a.performed.has_produced.has_title
from	Person a
where	a.name = "Picasso"

Figure 5: Query $Q1$

We illustrate query rewriting with the mapping shown in

Fig. 4. The system needs to rewrite the original query into one or more queries that find all paintings of Picasso in source *http://www.art.com*. The resulting queries are XML queries that use the XPath location paths defined in the mapping rules.

Basically, a user query ranges over a set of schema paths that have to be decomposed into the schema paths of the mapping rules. For example, query $Q1$ in Fig. 5 is composed of three paths : Person, name and performed.has_produced.has_title. The algorithm presented in Section 5 tries to match each query path with concatenations of rule schema paths. For example query path performed.has_produced.has_title is obtained by concatenating the schema paths of rules R_3 and R_4. By replacing the schema paths by the corresponding concatenations of source paths we obtain the rewriting illustrated in Fig. 6.

select	a/ARTIFACT/TITLE
from	http://www.art.com//ARTIST a
where	a/NAME = "Picasso"

Figure 6: Query $QS1$

Query $QS1$ can then be translated into an executable XML query (e.g. the Quilt query of Fig. 7). Query $QS1'$ is evaluated by the mediator on the XML document obtained by the URL in the *document* function-call of the query. In most cases, the size of this document is very important compared to the final query result (the titles of Picasso's artifacts only represent a small part of the document *http://www.art.com*) and a better solution would be to filter as much data as possible already at the source level before forwarding it to the mediator. Such a filtering is possible if the source is XPath enabled. For example, query $QS1''$ in Fig. 8 returns the same result as the previous one, but is completely evaluated at the source level and the mediator only has to forward the result to the user.

FOR	$\$a$ **IN** document("http://www.art.com")//ARTIST
WHERE	$\$a$/NAME='Picasso'
RETURN	$\$a$/ARTIFACT/TITLE

Figure 7: Query $QS1'$

Observe that a "complete" rewriting of our tree queries into XPath is not always possible and some more query processing might be necessary at the mediator level. This issue will be described in more detail in Section 6.

3 Related Work

C-Web is based on the standard model for query mediation : users formulate queries in terms of the ontology (mediation schema) and the portal (mediator) translates these queries and the obtained results according to the mapping rules and the source query facilities. Query mediation has been extensively studied in the literature for different kinds of mediation models and for various source capabilities. Tsimmis [20], YAT [7], Infomaster [14], Information Manifold [19], Tukwila [21, 17] and PICSEL [15] are among the most prominent examples of mediation systems. Our approach is closely related to the last three of them which follow the *local as view* approach, where sources are defined (independently of other sources) as relational views on the mediator schema.

In our case, sources are described by simple mapping rules relating XML fragments to an ontology of concepts connected by roles and organized in a concept/subconcept hierarchy. More precisely, a mapping rule is defined as a couple of paths P and Q where P is a standard XPath location path and Q is a path in the ontology. The definition of abstract views for XML documents as described in [10] is a similar approach adapted for large-scale XML repositories. This approach differs from ours in two points : (i) the mapping rules do not use variables in their definition and (ii) query translation is based on an efficient bottom-up rewritting algorithm.

Other query rewriting algorithms [18] are based on efficient implementations for evaluating query subsumption and satisfiability. Among these implementations, our rewriting algorithm might be best compared to the MiniCon algorithm [21] which improves the bucket algorithm [19] by considering the interaction of variables with the available views (rules) during query rewriting (instead of generating rewritings (buckets) for each subgoal independently before combining them into final query rewritings).

Our approach considers XPath [8, 24] enabled XML sources. XPath is a *tree pattern language* which allows to characterize XML fragments according to their position in the document tree, their type and their contents. Whereas XPath does not have the full expressive power of XML query languages, the choice of using XPath as part of a mapping language for XML documents is interesting for several reasons. First, XPath is already part of other XML-related languages [1] for the *transformation* (XSLT [12]), *linkage* (XLink [11]) and *querying* (XQL [22], XQuery [5] and Quilt [6]) of XML documents. Second, XPath is already implemented and used in a variety of tools, for example, for extending a standard Web server into an XPath server[3]. Finally, it is used by an important number of XML developers who do not have to learn a new language for writing mapping rules.

4 A Path Mapping Language for XML

Mapping rules describe XML resources by associating XPath location paths to paths in the conceptual schema of an ontology, denoted in the following as *schema paths*. Given a set of sources, the rules can be viewed as defining a virtual database that conforms to the conceptual schema, populated by XML fragments and relationships between them. This provides the basis for answering queries posed on the conceptual schema by appropriate sets of XML fragments.

4.1 XPath Location Paths

We rely on XPath-enabled XML resources to uniformly query and retrieve (fragments of) XML documents. The

[3] See for example *fragserver*, http://www.xml.com/pub/r/676.

FOR $\$a$ **IN** document("http://www.art.com//ARTIST[NAME='Picasso']//-
ARTIFACT/TITLE")
RETURN $\$a$

Figure 8: Query $QS1''$

XML Path Language (XPath) [8] is a W3C Recommendation for addressing parts of an XML document using *location paths*. A location path is a sequence of location steps. Each location step is evaluated in some *context*, which is a set of XML nodes (elements or attributes) defined by the previous location step. In general, the context of the first location step is defined by the root node of the XML document, specified by a URL. Location steps can be decomposed into three parts:

1. an *axis*, which specifies the structural relationship (child, descendant, ancestor, attribute etc.) between the nodes selected by the location step and the context node,

2. a *node test*, which specifies the node type (node, processing instruction, comment, text) and the *expanded name* of the nodes selected by the location step, and

3. *optional predicates*, which use arbitrary expressions to further refine the set of nodes selected by the location step.

Location paths can be used in different ways. First, a location path can be considered as a pattern for *selecting* XML fragments. For example, location path *http://www.art.com/descendant::ARTIST* selects all XML elements of type `ARTIST` which are descendants of the root element of the XML documents in *http://www.art.com*. Here, *http://www.art.com* is a URL defining the context for location step *descendant::ARTIST*, in which *descendant* is an axis and *ARTIST* is a node test. Second, location paths can be considered as a way to *navigate* from some node to other nodes in the document. In general, an axis may specify not only a step *down* from a node, as in the child and descendant axes illustrated above, but also sideways (axes *following/preceding/following-sibling/preceding-sibling*) and upwards steps (axes *parent/ancestor*).

Predicates : An important feature is the possibility to use location paths in predicates, since it allows to select nodes according to the properties of related nodes in the document tree. For example, predicate *[child::NATIONALITY = 'Spain' and child::ARTIFACT/child::TITLE = 'Guernica']* is true for all nodes with a child of type `NATIONALITY` with content 'Spain' and a child of type `ARTIFACT` with a child of type `TITLE` with content 'Guernica'.

Abbreviated syntax : Since *child* and *descendant* are the most frequently used axes, there also exists an *abbreviated syntax* which takes the child axis by default (if no axis is specified) and represents the descendant axis by a double-slash (//). For example, location path *child::A/descendant::B/child::C* can be abbreviated to */A//B/C* . In the following, we will use this abbreviated syntax whenever possible.

In this paper, we focus on location paths using only the child/descendant axis and *conjunctive* test predicates constructed on node (element/attribute) names and attribute values. The first restriction guarantees that each location path can be evaluated "locally" on any XML fragment of a document without considering the rest of the document. In Section 6, we will see that this constraint is useful for query processing with XPath enabled Web servers and could be discarded with the price of higher data communication. The second restriction reduces the complexity of the query rewriting algorithm as described in Section 5.

4.2 Ontologies and Schema Paths

An *ontology* is a 5-tuple $\mathcal{O} = (C, R, source, target, isa)$, where (i) C is a set of concepts, (ii) R is a set of binary roles, (iii) $source$ and $target$ map roles to their domain and target in C, respectively, and (iv) isa is an inheritance relationship between concepts in C. We assume the usual properties of isa. In particular it defines a hierarchy on C. We also consider ontologies to be symmetric : each role $r \in R$ has an inverse role, denoted r^-, in R (in Fig. 3 inverse roles are within parentheses). Obviously, $target(r^-) = source(r)$, and $source(r^-) = target(r)$.

The semantics of an ontology is defined by the databases DB that conform to it : DB contains a set of objects (instances) for each concept in C. These objects are related by instances of roles in R, which satisfy the typing constraints implied by $source$ and $target$. Roles are multi-valued, i.e. any instance of concept $source(r)$ can be related to zero or more instances of concept $target(r)$ by role r. The isa component of $\mathcal{O}$ is interpreted as subset relationship and role inheritance. Namely, if $c\ isa\ c'$, then the set of objects of c is a subset of the set of objects of c' and all roles defined between some concepts c and c'' are also defined between all subconcepts of c and c'' respectively. We say that c, c' are *isa-related* if either $c = c'$, $c\ isa\ c'$ or $c'\ isa\ c$. We also define $low(c, c')$ to denote the concept that is lower in the isa hierarchy as follows : $low(c, c') = c$ if either $c = c'$ or $c\ isa\ c'$. We also assume that, in general, all pairs of concepts that are not isa-related are disjoint.

Role paths and derived roles : A *role path* of length n ($n \geq 1$) is a sequence $r = r_1 \ \ldots \ r_n$, where r_i are roles, such that for all $1 \leq i < n$, $target(r_i)$ and $source(r_{i+1})$ are isa-related. The source and target of a role path are defined by the source and the target of its extremities : $source(r) = source(r_1)$ and $target(r) = target(r_n)$. Clearly, the composition of a role path r and a role path r', denoted $r \circ r'$, is well-defined provided that $target(r)$ and $source(r')$ are isa-related.

From the definition it results that all roles in R are also role paths of length 1. For example, role *performed* is a role path of length 1 with source Actor and target Activity. A role path r of length > 1 defines a *derived role* denoted by $role(r)$, from instances of its source concept to instances of its target concept. For example, role path *technique.used_material* defines a derived role, $role(technique.used\_material)$, between concept Man Made Object and concept Material.
To understand the second constraint in the definition of role paths, let us consider role paths of length two. A sequence $r_1.r_2$ can be viewed as a derived role whose every instance connects an instance o of $source(r_1)$ with an instance o' of $target(r_2)$, through an intermediary o'' that must be an instance of both $target(r_1)$ and $source(r_2)$ which is only possible if $target(r_1)$ and $source(r_2)$ are *isa*-related.

Concept paths and virtual concepts : A *concept path* p is either of the form c, or a sequence $c.r$, where c is a concept and r is a role path, such that $source(r)$ and c are isa-related. The length of p is 0 in the first case, and the length of the role path r in the second case. The $source$ and $target$ of concept path c is c itself. The source and target of $p = c.r$ are defined as: $source(p) = c$ and $target(p) = target(r)$. The composition of a concept path p and a role path r, denoted $p \circ r$, is well-defined provided that $target(p)$ and $source(r)$ are isa-related.
A concept path $p = c.r$ can be viewed as defining a *virtual concept*, standing for "the instances of $target(p)$ that can be reached from $source(p)$ by following the roles in p, in order". We denote the virtual concept defined by a concept path p by $conc(p)$. Given a database that conforms to $\mathcal{O}$, extents of virtual concepts are uniquely-defined.
Observe that concepts can only appear at the beginning of a concept path and it is not possible to restrict derived roles to subconcepts. For example, we could specialize a concept path $p = c.r_1.r_2$ by introducing a subconcept c' of $low(target(r_1), source(r_2))$: $p' = c.r_1.c'.r_2$. The virtual concept $conc(p')$ defined by path p' would obviously be a subconcept of the virtual concept $conc(p)$ defined by path p (in a similar way, this holds for derived roles with intermediate concepts). Whereas this introduces expressive power to the model (mapping rules and queries can be more "precise") it complicates not only definitions and but also query rewriting. In this paper, for the sake of simplicity and clarity, we present a simple model where intermediate concepts are not considered neither in concept paths nor in derived roles.

View ontologies : Given an ontology $\mathcal{O}$, a set of role paths $\mathcal{R}$ and a set of concept paths $\mathcal{P}$ in $\mathcal{O}$, we can define a *view ontology* $\mathcal{O}_V = (C_V, R_V, source_V, target_V, isa_V)$. The set of concepts C_V is initialized by the set of all virtual concepts defined by concept paths in $\mathcal{P}$. As in the original ontology, the sub/superconcept relationship isa_V has subset semantics. Clearly, given a database that conforms to $\mathcal{O}$, the extent of $conc(p)$ is a subset of the extent of $target(p)$ and all of its superconcepts in $\mathcal{O}$. Hence, by extension we say that $conc(p)$ is a subconcept of $target(p)$. In the sequel, we also define concept inheritance between virtual concepts and their "suffix". A *suffix* of a concept path p is obtained by removing a prefix, and adding an appropriate concept in the beginning. For example, if $p = c_1.r_1.r_2.r_3$, then $source(r_3).r_3$ and $source(r_2).r_2.r_3$ are suffixes of p. It can be seen that the extent of p is a subset of the extent any of its proper suffixes (and consequently a sub-concept thereof).
R_V is the set of derived roles, defined by the role paths in $\mathcal{R}$. Let p be a role path in $\mathcal{R}$. Then, $role(p)$ is a derived role in $\mathcal{O}_V$ between $source(p)$ and $target(p)$. Similarly, extents (i.e. sets of pairs) are defined for the roles of $\mathcal{O}_V$. Thus, a database that conforms to $\mathcal{O}$ induces in a natural manner a database that conforms to $\mathcal{O}_V$.

4.3 Mapping Rules

Let V be a set of variables, and U be a set of URLs. A *mapping rule* is an expression of the form $R : a/q\ [as\ v] \leftarrow p$, where

- R is the rule's *label*,
- $a \in V \cup U$, the rule's *root*, is either a variable or a URL,
- q is an XPath location path,
- $[as\ v]$ is an optional *binding* of v, where $v \in V$ is a variable, and
- p is a schema path : more precisely, p is a role path if a is a variable and a concept path otherwise.

Rule R is called a *relative mapping rule* if its root is a variable v, and an *absolute mapping rule* otherwise. In the first case, v is the *root variable* of R, and this occurrence of v is a *use* of the variable. If the optional component $as\ v$ occurs in R, then the rule is called a *binding rule* for variable v. Let $lp(R), sp(R)$ denote R's location path and schema path, respectively.
Given a set of mapping rules, we define *reachability* for rules and variables, as follows: Each rule whose root is a URL or a reachable variable is reachable, and each variable bound in a reachable rule is reachable. A *mapping* M over $\mathcal{O}$ is a set of mapping rules such that 1) labels are unique (that is, no two rules have the same label), 2) all rules are reachable (hence so are all variables) and 3) the concepts and roles used in its rules occur in $\mathcal{O}$.
Note that the binding-use relationships between variables in a mapping may be cyclic. The simplest case of a cycle is a rule whose left-hand-side contains $v/A\ as\ v$ (provided that v can be reached from a URL by other rules). A mapping is *cyclic* if it contains a binding-use cycle.
Although we have allowed the binding clause $as\ v$ in rules to be optional, for many of the definitions in the sequel, it is convenient that each rule has such a clause, and from now we assume that this is the case. Obviously, new (and distinct) dummy variables can be added to M, to satisfy this requirement. However, this assumption is purely for presentation purposes. A designer of a mapping may include them only when they are useful for the design.

Concatenation of mapping rules : Two rules $R_1 : a/q_1\ as\ v_1 \leftarrow p_1$, $R_2 : v_1/q_2\ [as\ v_2] \leftarrow p_2$, can be *concatenated*, if the composition of their schema paths, $p_1 \circ p_2$

is well defined[4]. Note the constraint that the root of R_2 is bound in R_1 and that concatenation is possible only if p_2 is a role path. The result of the concatenation is the rule $R_1.R_2 : a/q_1/q_2\ [as\ v_2] \leftarrow p_1 \circ p_2$.
Given a mapping M, its *closure* is the set of all rules that can be obtained from M by repeated concatenation. It is denoted by M^*. Its *expansion*, denoted $\hat{M}$, is the set of absolute rules in M^* ($\hat{M} \subseteq M^*$). Note that if M is cyclic, then M^* and $\hat{M}$ are infinite, and these sets can be computed by a bottom-up fixpoint computation otherwise (when M is acyclic).

Finite representation of cyclic mappings : Although M itself can be viewed as a finite representation of M^* and $\hat{M}$, when M has cycles, the following alternative representation is useful in the sequel. Recall that a rule of M^* is the concatenation of a sequence of rules of M. Since each of these binds a variable, there is a sequence of variables associated with intermediate points in the concatenated rules. For example, if R_1 binds v_1, R_2 binds v_2, and R_3 also binds v_1, then if $R_1.R_2.R_3$ is defined, it has a cycle, since it visits v_1 twice. For a relative rule, there is also a variable associated with its starting point. We call a rule of M^* (absolute or relative) *acyclic* if it has no cycle, i.e. it does not visit the same variable twice. We call a relative rule of M^* a *minimal cycle for variable* v if its root as well as bound variable is v and it does not occur in between and no other variable occurs in it more than once.
Obviously, if M is acyclic, then all rules in M^* are acyclic, and the set of minimal cycles is empty. If M is cyclic, then the closure of the set of acyclic rules in M is a finite and proper subset of M^*. Now, consider a rule $R = R_1.R_2$ of M^*, where $R_i, i = 1, 2$ are sequences of rules in M, and assume the bound variable of R_1 is v. If R' is a minimal cycle for v, then R can be *expanded* to a new rule, by inserting R' between R_1 and R_2, provided that the concept paths of R_1, R', R_2 can be composed. It is easy to see that any rule of M^* can be obtained from acyclic rules by repeated expansions. Thus, the collections of acyclic rules and minimal cycles of M^* are a finite representation of M^* with respect to expansion (the information about which short rule or minimal cycle can be expanded at a given point with which minimal cycle can also be finitely represented).

Interpretation of mapping rules : Given an ontology $\mathcal{O}$, a mapping M over $\mathcal{O}$ defines a *view ontology* $\mathcal{O}_M = (C_M, R_M, source_M, target_M, isa_M)$ where C_M is defined by the concept paths of absolute rules in $\hat{M}$ and R_M is the set of derived roles of relative rules in M^*. For example, rule $R_1.R_3$ defines a virtual concept, $conc(Person.performed.has\_produced)$, and rule R_3 defines a derived role, $role(performed.has\_produced)$, between concept Person and concept Man Made Object. Clearly, if M has cycles, then $\mathcal{O}_M$ has an infinite set of concepts and roles.
A mapping M allows us to view a collection of fragments of XML documents reachable from the URLs in U as a database that conforms to $\mathcal{O}_M$. To define this database, the population of each virtual concept, $conc(p)$, is defined as the union of the set of fragments "returned" by all absolute rules R in $\hat{M}$ where $sp(R) = p$ or p is a suffix of $sp(R)$. The set of fragments X_R returned by some absolute rule R is defined inductively as follows. Initially, X_R is empty for all absolute rules R. We start with the set of absolute rules in M. The root of an absolute rule R is a URL u and X_R is assigned the set of XML fragments that can be obtained by applying the location path $lp(R)$ to the XML document identified by u. Now, we repeat the following step until no further changes occur : select an absolute rule R and a relative rule $R' : a/q\ as\ v \leftarrow p$ in M such that the concatenation $R.R'$ is defined and add to $X_{R.R'}$ all the XML fragments that can be reached from the fragments in the current value of X_R by the location path $lp(R')$.
Similarly, the relative rules of M^* are interpreted as roles of $\mathcal{O}_M$ in this database of XML fragments, represented by location paths. For an absolute rule R and a relative rule $R' : a/q\ as\ v \leftarrow p$ in M such that the concatenation $R.R'$ is defined, $lp(R')$ represents the role of $sp(R')$ between the extent of $conc(sp(R))$ and that of $conc(sp(R.R'))$.
Before leaving this subject, we note that as mapping rules may be added to a mapping, for example when new sources are added, the XML extents defined as above for the concepts can be viewed not as the full extents, but rather as subsets of the real (but unknown) extents. However, since query evaluation may only use currently known information, this has no real impact on it.

5 Tree Queries and Query Rewriting

The user views the community-web portal as a single database of fragments without knowledge of the source on which each fragment is located. We might then consider each fragment as an object whose identity is the location path of the fragment. Given a mapping M, we have seen above that it allows us to organize XML fragments into collections of instances of concepts $\mathcal{O}_M$, and also to view certain XML paths as representing roles of $\mathcal{O}_M$. It follows that one can in principle query this database of fragments using a query language, such as OQL, the standard for querying object databases. The answer of a query is defined in the standard manner, using the semantics of queries on object bases.
However, even though the answer is well-defined, an efficient evaluation requires that we use effectively the mapping M to translate the query into one or more queries on the sources. Our ability to do so may depend on the form of the query. This subject is taken up in the next section where we introduce a query language for C-Web portals and discuss how mapping rules can be used in the evaluation of queries expressed in this language.

5.1 Tree Queries

While a key-word based search is simple to use, it has a limited expressive power. On the other hand, a full-fledged query language such as OQL is powerful but may prove to be too complex for many users. We present a *tree query language* as an intermediary solution that is easier to use, yet sufficiently powerful for most needs.

[4] We do not define any restriction on the concatenation of location paths (rule left-hand-sides).

The *tree query language* is based on an OQL-like syntax with **select-from-where** clauses on schema paths. For example, the query $Q2$ in Fig. 9 finds the name of the person that has produced a man-made object with title "Mona Lisa".

select	b
from	Person a,
	a.has_name b
	a.performed.has_produced.has_title c
where	c = "Mona Lisa"

Figure 9: Query $Q2$

More formally, a *tree query* Q is an expression of the form :

Q:	**select**	x_i, x_j, ...
	from	$e_1\ x_1$,
		$e_2\ x_2$,
		...
		$e_i\ x_i$,
		...
	where	c_0 **and** c_1 **and** ...

where the x_i's are variables and each e_i in the **from** clause is either (1) a concept path defining a virtual concept (of the form c_i or $c_i.r_i$, where c_i is a concept and r_i is a role path) or (2) a variable x_i followed by a role path r_i (of the form $x_j.r_i$). In the first case c_i/$c_i.r_i$, and in the second case r_i, are called the *binding path* of x_i, denoted $bp(x_i)$. We distinguish between the two cases since in the first e_i is a concept (possibly virtual) over whose extent x_i ranges; in the second case, x_i ranges over the concept defined by traversing the role r_i from the concept of x_j.
No restructuring is allowed in the **select** clause. Although this may add expressive power to the language, we feel it is not strictly needed for our application. Furthermore, restructuring is performed at the integration site, hence is orthogonal to the issue of retrieving data from sources, addressed in this paper.
The **where** clause is a conjunction of simple predicates, where a simple predicate is of the form $c_k \equiv x_i \theta d$ in which $\theta \in \{=, <, >, \leq, \geq\}$ and d is an atomic value. Thus, it is not possible to express joins by equalities between variables, i.e., by predicates of the form $x_i = x_j$. This restricts the expressive power of the query language but simplifies the rewriting and evaluation of queries.
Note that schema paths appear exclusively in the **from** clause. They do not appear in the **select** clause nor in the **where** clause of a query. This syntax simplifies the presentation of our rewriting algorithm. It is easy to show that a query with schema paths in the **select** and the **where** clause can be rewritten into an equivalent query in which they appear only in the **from** clause.
Finally, the language has no quantifiers, aggregates, or subqueries. But, a variable x_j present in the **from** clause but not in the **select** or **where** clauses, is implicitly existentially quantified. Thus, queries with certain kinds of existential quantification can be translated to the above form.

Tree representation of tree queries : We assume the existence of a partial order $<$ on the variables, such that if $x_j.r_i\ x_i$ occurs in the **from** clause, then $x_j < x_i$. Thus, the **from** clause can be represented as a forest, with the variables as nodes, and an edge connecting parent x_j to child x_i if $x_j.r_i\ x_i$ occurs in the clause. In addition, since no joins are allowed in the **where** clause, a query can be represented by a forest and decomposed into a cross product of several queries each of which is represented by a tree. Therefore, in the following and without loss of generality, we restrict attention to tree queries with exactly one variable bound by a schema path.
We represent a query Q as a labeled tree $T(Q) = (X, par, bp, ops)$ where a node is labeled by a variable in X (the set of variables in Q), par is the parent binary relation between nodes defined by the partial order on variables, bp maps the node to the variable binding path and *ops* maps the node to a set of operations :

- for each variable x, add $\exists$ to $ops(x)$,
- for each variable x in the **select** clause of q, add π to the set of operations $ops(x)$,
- for each predicate $x\theta d$ in the **where** clause, add $\sigma_{x\theta d}$ to $ops(x)$.

From the above definition, it results that if a variable x occurs *only* in the **from** clause, then $ops(x) = \{\exists\}$.

5.2 Query Rewriting Algorithm

Tree queries are evaluated on the database induced by some mapping M. In the following, we present a binding algorithm that binds variables in a tree query Q to rules in M^*. The result is a set of such variable/rule bindings that can be used to rewrite tree queries into standard XML queries using XPath location paths for instantiating variables. Query evaluation is described in Section 6.

Binding variables to rules : Take, for example, query $Q2$ in Fig. 9 and the mapping shown in Fig. 4. Intuitively, rule R_1 might be used to find persons for variable a, rule R_2 can be used to find the names of these persons for variable b, and the concatenation of rules $R_3.R_4$ can be used to find values for the variable c. The set of associations $\{a \mapsto R_1, b \mapsto R_2, c \mapsto R_3.R_4\}$ provides the information about which rules from the closure M^*[5] can be used for translating the query to queries on the XML sources.
First, let us introduce the notion of prefix of a tree T. We say that a tree T' is a *prefix* of a tree T if its set of nodes is a subset of the set of nodes of T, its set of edges is the restriction of T's set of edges to that subset, and its root is the same as that of T. Note that it follows from the definition that if T' contains a non-root node of T, then it contains the nodes and edges up to T's root. Formally, a *variable to rule binding*, or shortly *variable binding*, for a query Q is a mapping β on a set, denoted $dom(\beta)$, that is either empty, or is the set of nodes of a prefix of $T(Q)$. A variable binding is *full* if it

[5] Recall that M^* contains all concatenations of rules in M.

is defined on all variables of Q, and *partial* otherwise. The *empty* binding is denoted β_ϕ.
If $dom(\beta)$ is not empty, then β associates each variable in it with a rule of M^*, such that the following holds:

1. if x is the root of query Q, then $\beta(x)$ is an absolute mapping rule such that the query binding path $bp(x)$ of variable x denotes a superconcept of $conc(sp(\beta(x)))$ in the view ontology $\mathcal{O}_M$, i.e. $conc(sp(\beta(x)))\ isa_M\ conc(bp(x))$.
2. else, let $par(x) = x'$, then
   - the root variable of rule $\beta(x)$ is bound in rule $\beta(x')$,
   - the role path of rule $\beta(x)$ is equal to the role path $bp(x)$ and
   - the composition of the role paths of the rules $\beta(x')$ and $\beta(x)$ is well-defined, hence the concatenation of the two rules is well-defined.

Regarding the first case, if x is the root of Q, then it is bound to some, possibly virtual concept by its binding path $bp(x)$, that has the form c or $c.r$. An absolute rule can provide instances for this concept if its schema path, viewed as a virtual concept, is a sub-concept of c or $c.r$. In the first case, c is a concept of $\mathcal{O}$ that is a superconcept of $conc(sp(\beta(x)))$. In the second case $c.r$ is a virtual concept, and it is a suffix of or equal to $sp(\beta(x))$. For the second case, the assumption that if β is defined on x then it is defined on the parent of x follows from the requirement that its domain is a prefix of $T(Q)$. In this case, the declaration of x in Q has the form $x'.q\ x$, and $bp(x) = q$. Answers for x can be obtained from answers for x', by following the binding path q of x.
For example, for query $Q2$ in Fig. 9, the query root variable a is bound in the query to the (real) concept Person. In this case, rule $R1$ with schema path $sp(R_1)$=Person provides a binding for a. The binding path of variable b in $Q2$ is *has_name*, so we need a relative rule whose schema path uses the variable bound by rule R_1 and defines the role *has_name*; R_2 is such a rule. Similar reasoning applies to the binding for variable c.
In query $Q3$ illustrated in Fig. 10, the query root variable a is bound to a virtual concept. Instances for it may be found from absolute rules that have the binding path of a as a suffix; $R_1.R_3$ with schema path Person.performed.has_produced is such a rule.

select	b
from	Activity.has_produced a,
	a.consists_of b,
	a.has_title c
where	c= "Mona Lisa"

Figure 10: Query $Q3$

Given a query, we need to find all the full variable bindings. Indeed, each such binding provides, as shown below, a subset of the answer. By taking the union of all these answers, we have a maximal answer, with respect to the given sources and the given mapping.

Calculating variable bindings : A general algorithm for finding variable bindings is sketched in the following. It starts with the unique empty partial binding. Then it enters a loop. In each pass through the loop, it selects a partial binding on i variables, and extends it to one more variable, using some rule of M^*[6]. Note that to extend the empty binding β_ϕ, we need to use an absolute rule of M^*, whereas to extend a non-empty binding we use a relative rule of M^*. In the latter case, the binding can be extended to a new variable only if it is already defined for its parent.

Variable binding algorithm (sketch) :

Input:	query Q, with root variable x_1;
	mapping M;
Output:	a set of variable bindings;
Algorithm:	initialization: let $B = \{\beta_\phi\}$;
	loop: while B changes do $\{$
	select a partial binding β from B
	and a rule R in M^*
	such that R extends β to β'
	let $B = B \cup \{\beta'\}$
	$\}$
Result:	output the full bindings in B.

As an example, we will create the set of variable bindings B for query $Q2$ of Fig. 9 in three steps. The algorithm starts with the singleton $B = \{\beta_\emptyset\}$ containing the empty variable binding. The first step extends the empty binding with binding $\beta_1 = \{a \mapsto R_1\}$: Rule R_1 is the only (absolute) rule in M^* with schema path $sp(R_1)$ the binding path of variable a is a suffix of. The second step extends binding β_1 by binding rule R_2 to variable b. Then, $\beta_2 = \beta_1 \cup \{b \mapsto R_2\}$. The third step finally results in binding $\beta_3 = \beta_2 \cup \{c \mapsto R_3.R_4\}$ and $B = \{\beta_\emptyset, \beta_1, \beta_2, \beta_3\}$ is the binding set produced by the above algorithm for query $Q2$ and mapping M in Fig 4. The algorithm returns the only full variable binding β_3 in B (obviously the algorithm might generate more than one full variable bindings).
An obvious problem with this algorithm is that it may not terminate. When M contains cycles, M^* is infinite, and rules of M^* are represented by arbitrary long concatenable sequences of rules of M.

Acyclic Mappings : Let us temporarily assume that M is acyclic. The algorithm can be optimized in several ways. First, since M^* is finite and it is rather stable, hence used for many queries, it can be computed in advance. We expect M^* to be of tractable size in most practical cases, but of course we incur the risk of a degenerated case where it is not. In the following we assume the precomputation of M^*. Recall that for each rule in M^*, its root is a variable or a URL, and it binds some variable.
An obvious improvement of the previous general naive algorithm is to compute the partial bindings using each one just once to compute all its extensions. For that, we choose

[6] The rules in M^* are sequences of rules of M.

Input:	the sequence of variables of query Q, in pre-order: $x_1, \ldots, x_n$;
	the set of mapping rules M^*;
Output:	a set of full variable bindings;
Algorithm:	initialization: let $B_0 = \{\beta_\phi\}$; $B_i = \phi, i = 1, \ldots, n$

loop: for each absolute rule R of M^*, if $sp(R)$ is a sub-concept of $bp(x_1)$
    then $B_1 := B_1 \cup \{x_1 \mapsto R\}$
loop: for $i = 2, \ldots, n$ {
    loop: for each binding β from B_{i-1} and rule R in M^* {
        if R's root is bound by the rule associated with x_i's parent, say y in β,
            and $sp(R) = bp(x_i)$,
            and the composition of $sp(\beta(y))$ and $sp(R)$ is well-defined,
        then $B_i := B_i \cup \{\beta \cup \{x_i \mapsto R\}\}$
    }
    now B_{i-1} can be discarded
}

Result: the set B_n.

Figure 11: **Variable binding algorithm**

for the variables of the query tree any order in which the root is first, and every other node occurs after its parent. This ensures that when we try to extend a binding to a variable, it is already defined on its parent. Without loss of generality, let us assume that the variables of the tree are arranged in pre-order: $x_1, \ldots, x_n$. A binding is represented as a vector of associations of variables to rules, in that order, namely $\{x_1 \mapsto R_1, \ldots x_n \mapsto R_n\}$. Here is the new version of the algorithm, illustrated in Fig. 11.

In the first step, we extend the empty binding to the root variable x_1. For each absolute rule R in M^* such that $sp(R)$ is a subconcept of the binding path $bp(x_1)$[7], we create a binding $\{x_1 \mapsto R\}$. Then, we iterate through the sequence of variables, from the left. Assume we have last visited variable x_{i-1} ($i > 1$), and have constructed a set of partial bindings that are defined on all variables up to and including x_{i-1}. Let x_i be the next variable, and let y be its parent. Necessarily, all the bindings we have constructed are defined on y. For each binding β, assume it associates rule R' with y, and that variable v is bound in R'. Then, for each relative rule R of M^* whose root is v, if the schema paths of R' and R can be composed, we extend β by $x_i \mapsto R$. Note that the edge from y to x_i is 'traversed in this step, and only in this step'. After all bindings that are defined up to and including x_i are computed, all previous partial bindings that are not defined on x_i can be dropped.

It is easy to prove that if β is a full variable binding such that the conditions defined in Section 5.2 hold, then the restriction of β to the prefix $x_1, \ldots, x_k$ ($0 \leq k < n$) is in B_k and β is in B_n. It is also obvious that each member of B_n produced by the algorithm satisfies conditions in Section 5.2, hence is a legal binding. Thus, the algorithm produces precisely the set of legal bindings.

Denote the maximum length of $bp(x_i)$ by $\mid bp(x_i) \mid$. We observe that part of the condition for extending a binding β with $\{x_i \mapsto R\}$, where x_i is not the root variable, and where R is a relative rule of M^*, is that $bp(x_i) = sp(R)$. Thus, all we need to do for this case is to compute all relative rules of M^* for which the length of the role path is limited by $\mid bp(x_i) \mid$. Recall that each relative rule is a concatenable sequence of rules of M, and that relative rules of M have schema paths longer than zero. It follows that $\mid bp(x_i) \mid$ is an upper bound to the length of sequences we need to consider. Thus, in the algorithm above, rather than using all relative rules of M^* in the loop, we use only those whose schema path length is limited by $\mid bp(x_i) \mid$ and, as far as relative rules are concerned, we need not worry about whether M is cyclic.

Cyclic Mappings : Now, consider the case that M^* is cyclic, and the case of the root variable x_1. Here, an absolute rule R in M^* needs to satisfy the condition that $bp(x_1)$ is a suffix of $sp(R)$ ($sp(R)$ defines a sub-concept of ($bp(x_1)$)). Now, such a rule is the concatenation of one absolute rule R' of M, with a (possibly empty) sequence R'' of relative rules in M. In the first case (R'' is empty), R' is a candidate rule for x_1. In the second case, R'' is a relative rule in M^*. Then let us first consider all relative rules R'' of M^* (possibly with cycles), where the length of their schema path is at most that of $bp(x_1)$ (it is obvious that the set of such rules is finite). For each such rule R'', we try to match $sp(R'')$, *from the right*, with $bp(x_1)$. There are two cases to consider:

1. $sp(R'')$ matches a suffix of $bp(x_1)$, i.e. $bp(x_1) = p.sp(R'')$ where p is a schema path of length > 0. Then, for each absolute rule R' of M such that p is a suffix of $sp(R')$, if $R'.R''$ is defined, then it is an absolute rule of M^* such that $bp(x_1)$ is a suffix of $sp(R'.R'')$.

2. $sp(R'')$ matches all of $bp(x_1)$, i.e. $bp(x_1)$ is a suffix of $sp(R'')$. Let R'''s root variable be v. Then for each absolute rule R' of M^* that binds v, if the concatenation $R'.R''$ is defined, then $bp(x_1)$ is also a suffix of $sp(R'.R'')$. Such a rule R' is necessarily obtained from an acyclic rule that binds v, by expanding it any number of times with minimal cycles.

[7] The (virtual) concept denoted by $conc(sp(R))$

It can be seen that all candidate rules for x_1 fall into one of the two cases. While we can provide a full enumeration of all the candidate rules that fall into the first case, in the second case, all we can do is to list the (finite) set of candidate acyclic rules, and the (finite) collection of minimal cycles (see Section 4.3 for the definition of minimal cycles). The latter can be pruned to *relevant* cycles, as follows : define a minimal cycle to be *relevant* if its root (and bound) variable occurs in a candidate acyclic rule, or in a relevant minimal cycle, and the composition of the relevant schema paths is well-defined.
The first step of the algorithm for the acyclic case can be easily adapted to provide all bindings that fall into the first case, and all bindings using the concatenation of an acyclic absolute rule with a relative rule whose derived role has length bound by $\mid bp(x_i) \mid (i > 1)$. Of course, we expect that when bindings that fall into the second case are found, query evaluation will need to deal with minimal cycles, and thus contain some form of a fixpoint computation.

6 Query Processing

6.1 Evaluating Tree Queries with Quilt

Let B be a set of full variable bindings calculated by the foregoing algorithm for some query Q and mapping M. Then each variable binding β in B can be used to create a new query QS where concept paths in Q are replaced by XPath location paths. More precisely, we can obtain a first rewriting of Q by replacing each concept path $bp(x)$ in Q by the corresponding XPath location path $lp(\beta(x))$. For example, for query $Q2$ in Fig. 9 and mapping M shown in Fig. 4, the previous binding algorithm returns a complete binding $\beta = \{a \mapsto R_1, b \mapsto R_2, c \mapsto R_3.R_4\}$ which can be applied to query $Q2$ by replacing each binding path $bp(a), bp(b)$ and $bp(c)$ by the corresponding location path $lp(\beta(a))$, $lp(\beta(b))$ and $lp(\beta(c))$. The obtained query is illustrated in Fig. 12.

select	b
from	http://www.art.com//ARTIST a,
	a/NAME b,
	a/ARTIFACT/TITLE c
where	c = “Mona Lisa”

Figure 12: Query $QS2$

This query can easily be expressed in any XML query language using XPath for binding variables such as Quilt [6], XQL [22] and XQuery [5]. For example, the following translation of tree query $QS2$ into a FLWR expression of the Quilt language is straightforward and is illustrated in Fig. 13.
After loading the root of the document reached from URL *http://www.art.com*, query $QS2'$ could be directly evaluated by a Quilt query engine (e.g. Kweelt [23]) at the integration (mediator) site. It is evident that this is not very efficient if the size of the document is large compared to the size of the final result. Indeed, if the source has some query capabilities, it might be possible to push some filtering to the source level. In the following, we assume XPath enabled sources.

FOR	$\$a$ **IN** document(“http://www.art.com”)//ARTIST,
	$\$b$ **IN** $\$a$/NAME,
	$\$c$ **IN** $\$a$/ARTIFACT/TITLE
WHERE	$\$c$ = “Mona Lisa”
RETURN	$\$b$

Figure 13: Query $QS2'$

Then, for example, if *http://www.art.com* is an XPath enabled web server, the rewriting of query $QS2'$ into query $QS2''$ of Fig. 15 can directly be evaluated by the source itself.
The source location path in query $QS2''$ (*http://www.art.com/ARTIST[ARTIFACT/TITLE='Mona Lisa']/NAME*) returns a set of XML fragments [16] which are bound consecutively to variable $\$b$ and forwarded to the user.
This illustrates that a query or at least a part of it can be evaluated by XPath enabled sources in order to minimize the volume of data exchanged between the source and the mediator. Because of the XPath particularities, it is not always possible to obtain a *complete* rewriting of the initial query into a *single* XPath expression. This is due to the fact that XPath is a pattern language for XML nodes, but cannot create new nodes (as it is possible in our tree query language by using projection). For example, the source query $QS3$ in Fig. 14 returns a set of couples (d, e) where d is the title and e is the material of an artifact created by Van Gogh (each couple corresponds to a new node with two children).

FOR	$\$a$ **IN** document(“http://www.art.com”)//ARTIST,
	$\$b$ **IN** $\$a$/NAME, $\$c$ **IN** $\$a$/ARTIFACT,
	$\$d$ **IN** $\$c$/TITLE, $\$e$ **IN** $\$c$/MATERIAL
WHERE	$\$b$ = 'Van Gogh'
RETURN	$\$d$, $\$e$

Figure 14: Query $QS3$

Whereas source *http://www.art.com* might return the title and material of each artifact “independently” by evaluating the two XPath patterns *//ARTIST[NAME='Van Gogh']/-ARTIFACT/TITLE* and *//ARTIST[NAME='Van Gogh']/-ARTIFACT/MATERIAL*, it is not possible to obtain the title and material of each artifact “together”. Nevertheless, the set of fragments returned by the source location path in $QS3$ might be reduced by pushing selections and existential quantification to this path and returning only the `ARTIFACT` elements instead of the artists. It is easy to see that, in order to obtain the final result, it is sufficient to project on `TITLE` and `MATERIAL` subelements of the `ARTIFACT` elements returned by the new document URL *http://www.art.com//ARTIST[NAME='Van Gogh']/ARTIFACT[TITLE and MATERIAL]*.

6.2 Query Decomposition

Given the restrictions enforced by XPath mentioned previously, our idea is to *decompose* the initial tree query into a query (XPath expression) to be evaluated at the source and a

```
   FOR     $b IN document("http://www.art.com//ARTIST[ARTIFACT/TITLE='Mona
                  Lisa']/NAME")
RETURN     $b
```

Figure 15: Query $QS2''$

query to be evaluated at the mediator. The objective of the following decomposition algorithm, is to do as much as possible of the evaluation at the source level. Our goal is to minimize the size of the data fetched by the mediator by creating a source (XPath) query that (1) verifies all selections and existential quantification in the original query and (2) returns the smallest fragments necessary for evaluating the "rest" of the original query at the mediator level.

Let $T(QS) = (X, par, lp, ops)$ be the tree representation (see Section 5) of a query QS where X is the set of variables and for each variable v, $par(v)$ is its parent variable, $lp(x)$ is the XPath location path (corresponding to some binding β), and $ops(v)$ is the set of operations defined on v.

Query Composition : Query QS_1 is called a *prefix* of query QS_2 (QS_2 is a *suffix* of QS_1) if they share exactly one variable x satisfying the following conditions : (1) x is the root node (variable) of QS_2, (2) x is bound by the empty XPath location path $self()$ in QS_2 ($lp_2(x) = self()$) and (3) x is a projection variable in QS_1 ($\pi \in ops_1(x)$). Variable x is called the *pivot* of QS_1 and QS_2. The *composition* of a prefix QS_1 with its suffix QS_2 is denoted by $QS_1 \circ QS_2$ and returns a new query QS where 1) the root variable of QS is the root variable of QS_1, 2) QS contains all nodes (variables) in QS_1 and QS_2 and 3) maintains for all variables the partial order, location paths and operations in QS_1 and QS_2 respectively. More precisely, for pivot (variable) x, query QS keeps the parent variable and location path of QS_1 and all operations defined in QS_1 and QS_2 ($ops(x) = ops_1(x) \cup ops_2(x)$).

For example, the composition of the two queries $QS3_a$ (prefix) and $QS3_b$ (suffix) illustrated in Fig. 16 is defined and results in query $QS3$ of Fig. 14.

```
   FOR     $a IN document("http://www.art.com")//ARTIST,
           $b IN $a/NAME, $c IN $a/ARTIFACT,
           $d' IN $c/TITLE, $e' IN $c/MATERIAL
 WHERE     $b = 'Van Gogh'
RETURN     $c
```

Query $QS3_a$

```
   FOR     $c IN self(),
           $d IN $c/TITLE, $e IN $c/MATERIAL
RETURN     $d, $e
```

Query $QS3_b$

Figure 16: Queries $QS3_a$ and $QS3_b$.

Query Equivalence : Queries QS and QS' are called *equivalent* iff they return the same result for any set of XML sources S. Given a query QS there might exist several equivalent tree queries QS'. For example, it is easy to see that, given a node (variable) v in QS one might add a new node (variable which is not in QS) with the same parent and location path as v and $ops(v') = \{\exists\}$ without changing the result of QS (note that all variables in a tree query are existentially quantified).

Query Decomposition : A sequence of tree queries (QS_0, QS_1), is a decomposition of QS if $(QS_0 \circ QS_1)$ is defined and *equivalent* to QS. For example, the sequence of queries $(QS3_a, QS3_b)$ shown in Fig. 16 is a decomposition of query $QS3$ illustrated in Fig. 14.

In the context of query evaluation, we consider decompositions where QS_0 is a prefix of QS_1. Then, from the definition of query composition, it follows that the root of QS_1 is a projected variable in prefix QS_0, i.e. the suffix subquery QS_1 can be evaluated on the result obtained by query QS_0 (this is true since we only allow the descendant/child axes in variable binding location paths). In other words, prefix QS_0 can be sent to the sources for evaluation, and the suffix query QS_1 can be evaluated at the mediator level on the fragments resulting from the evaluation of QS_0.

The problem becomes to find a decomposition of QS such that prefix QS_0 filters as much XML fragments as possible at the source level before the evaluation of the suffix query. It is evident that this decomposition process is restricted by the capabilities of the source query language, which in our case is XPath. As mentioned earlier, XPath only allows to project on one variable, i.e. QS_0 only contains one variable v where $\pi \in ops(v)$ and, as a consequence of the definition of query decomposition, the suffix query QS_1 shares the same variable v (its root).

6.3 Decomposition Algorithm

In the following, we describe a query decomposition algorithm for tree queries involving XPath expressions. Let QS be a tree query and u be the variable that corresponds to the root of the query. The decomposition algorithm starts with the trivial decomposition (QS_1, QS_2) where QS_1 corresponds to a projection on the location path of u ($X_1 = \{u\}$, $lp_1(u) = lp(u)$ and $ops_1(u) = \{\pi\}$) and QS_2 corresponds to the query obtained from QS by replacing the location path of u with $self()$ ($lp_2(u) = self()$). For example, for decomposing query $QS3$ of Fig. 14, the algorithm starts with two subqueries $QS3.a$ (prefix) and $QS3.b$ (suffix) illustrated in Fig. 17. It is easy to see that $(QS3.a, QS3.b)$ is a decomposition of $QS3$.

The decomposition algorithm is based on two simple observations. First, all existential quantifications defined by the **FOR** clause and all conditions in the **WHERE** clause can be "pushed" into the prefix query (source location path) without changing the result of the composition $QS_1 \circ QS_2$. However, the new prefix query is more "optimal" since it selects only

```
     FOR   $a IN document("http://www.art.com")//ARTIST
  RETURN   $a
```

Prefix Query QS3.a

```
     FOR   $a IN self(),
           $b IN $a/NAME, $c IN $a/ARTIFACT
           $d IN $c/TITLE, $e IN $c/MATERIAL
   WHERE   $b = 'Van Gogh'
  RETURN   $d, $e
```

Suffix Query QS3.b

Figure 17: Initial decomposition of $QS3$ to the *prefix* query $QS3.a$ and the *suffix* query $QS3.b$.

those fragments where the predicates on the source location path evaluate to true. Nevertheless, we would still need to load the whole document for evaluating projection. In fact, we would like to load, instead of the whole document, the smallest possible fragments that allow to evaluate all projections defined in the query. This can be obtained by "narrowing" projection to the smallest subtrees containing all projections of the original query (note that our tree queries only use the child/descendant axes and all instances of a variable v can be obtained from the instances of its ancestor variable). The following algorithm describes the previous discussion in three consecutive steps :

Step $\mathcal{S}1$: All variables in a tree query are existentially quantified, i.e. for all variable bindings v such that v **IN** u/q in the **FOR** clause of QS, there must exist at least one fragment that can be obtained from a fragment in u by evaluating location path q. This condition can be pushed into the prefix query by creating for each non-root variable v in QS a new existentially quantified variable v' ($ops_1(v') = \{exists\}$) in QS_1 called the *surrogate* of v and denoted $v' = surr(v)$. The partial order and location paths defined on the variables in QS are also maintained in QS_1 : $par(surr(v)) = surr(par(v))$ and $lp(surr(v)) = lp(v)$.

Step $\mathcal{S}2$: As for existential quantification, it is possible to copy all selection predicates in QS to the prefix query QS_1 : If $\sigma_{v\theta d} \in ops(v)$ then $ops(surr(v)) = ops(surr(v)) \cup \sigma_{surr(v)\theta d}$.

After applying steps $\mathcal{S}1$ and $\mathcal{S}2$ we obtain the new prefix query $QS3.a'$ illustrated in Fig. 18 (the suffix query $QS3.b$ has not been changed yet).

Step $\mathcal{S}3$: The prefix query should return the smallest possible fragments containing all information needed for query evaluation. This can be obtained by "narrowing" projection to the smallest subtrees containing all projections of QS. In our context, where the location paths in the query are formed using only the descendant and child axes, the solution is to project on the *least common* ancestor variable of all projection variables. More formally, let o be the least common ancestor of all projection variables (o is not necessarily a projected variable) in QS. Then, first we will narrow projection in QS_1 to $surr(o)$, i.e. move the projection from the root u of QS_1 to node $surr(o)$ such that $surr(o)$ will become the new pivot variable : $ops_1(u) = ops_1(u) - \{\pi\}$ and $ops_1(surr(o)) = ops_1(surr(o)) \cup \{\pi\}$. Second, variable o will become the new root of QS_2, i.e. we can remove all variables in QS_2 that are not descendants of o and $lp_2(o) = self()$. Observe that it is also possible to remove all direct subtrees (children) of o in QS_2 containing no projections. This is possible, since these subtrees only serve for filtering instances of o, which is already done by QS_1 after applying steps $\mathcal{S}1$ and $\mathcal{S}2$. Finally, in order to compose the new prefix and suffix, all occurrences of $surr(o)$ will be renamed into o in query QS_0.

After applying step $\mathcal{S}3$ and defining variable $\$c$, as the new pivot between $QS3.a'$ and $QS3.b$, we obtain the final prefix and suffix queries illustrated in Fig. 19 decomposing query $QS3$.

```
     FOR   $a IN document("http://www.art.com")//ARTIST
           $b' IN $a/NAME, $c' IN $a/ARTIFACT
           $d' IN $c'/TITLE, $e' IN $c'/MATERIAL
   WHERE   $b' = 'Van Gogh'
  RETURN   $a
```

Figure 18: *Prefix* query $QS3.a'$ obtained after application of steps $\mathcal{S}1$ and $\mathcal{S}2$.

```
     FOR   $a IN document("http://www.art.com")//ARTIST
           $b' IN $a/NAME, $c IN $a/ARTIFACT
           $d' IN $c/TITLE, $e' IN $c/MATERIAL
   WHERE   $b' = 'Van Gogh'
  RETURN   $c
```

Prefix query $QS3.a''$

```
     FOR   $c IN self()
           $d IN $c/TITLE, $e IN $c/MATERIAL
  RETURN   $d, $e
```

Suffix query $QS3.b''$

Figure 19: Final prefix ($QS3.a''$) and suffix ($QS3.b''$) queries for $QS3$.

Observe that steps $\mathcal{S}1$ and $\mathcal{S}2$ reduce the number of fragments returned by the source location path, whereas $\mathcal{S}3$ reduces the size of the fragments returned. Observe also that $\mathcal{S}3$ might increase the number of fragments but not the total size of returned data.

After these three steps, the prefix query $QS3.a''$ can be translated into a standard XPath location path

QS3':	**FOR**	$\$c$ **IN** document("http://www.art.com//ARTIST[NAME='Van Gogh']/-ARTIFACT[TITLE and MATERIAL]")
		$\$d$ **IN** $\$c$/TITLE, $\$e$ **IN** $\$c$/MATERIAL
	RETURN	$\$d$, $\$e$

Figure 20: Final query $QS3'$.

(*"http://www.art.com//ARTIST[NAME='Van Gogh']/-ARTIFACT[TITLE and MATERIAL]"*). Then, we replace the binding location path of the root variable of suffix query $QS3.b''$ with this path and the final query is illustrated in Fig. 20.

7 Conclusion and Future Work

In this paper, we have presented the integration of XPath-enabled XML resources according to high-level and domain specific ontologies. Source integration is obtained by mapping XPath location paths to conceptual paths in the ontology. Our path to path mappings can be exploited in several ways. In this paper, we have focused on the rewriting of tree queries. The presented rewriting algorithm has been added to our prototype which is implemented on top of Java and the object-oriented database system O_2 [13]) and OQL [9] as the query language.

In the general case, the query may ask for data from several sources. For example, it may ask for the biographical details of the painter of a given painting. We may find the name of the painter from one source, given the painting, and then use it to retrieve her biography from another. Obviously, this calls for a strategy that determines how to pass information from the results obtained from one source to queries sent to another source. We are currently studying a query composition algorithm that exploits partial variable-rule bindings for decomposing a query QS into a partially ordered set of XPath expressions (prefix queries) to be evaluated by the sources, where the partial order implies information passing, followed by an XML query (suffix) to be evaluated by the mediator on the results of these subqueries.

We are also studying a second application of mapping rules for the automatic creation of source descriptions [3]. Formally speaking, this corresponds to the materialization of the view ontology and the resulting C-Web database can be considered as a data warehouse for XML resources. One advantage of view materialization is that it avoids query rewriting and a standard query evaluator (e.g. SQL or OQL interpreter) can be used for query answering. A second advantage is that the C-Web repository contains a (partial) copy of the source information and a certain number of queries can be answered without accessing the XML resources (e.g. give me the URLs of the resources containing information about Picasso paintings). Nevertheless, in the presence of large collections of XML documents evolving with time, the storage and maintenance of materialized views becomes very costly, unless it is possible to control the size and the changes of the materialized information. Size control can be user driven, for example, by preloading descriptions corresponding to a query (partial view materialization) defined by the user. Change control can be source-driven, for example, by triggering the creation (update) of descriptions upon a change in the document contents.

Another important issue appears with the combined usage of 1) query rewriting, 2) automatically generated descriptions and 3) user defined descriptions during query evaluation. For example, a C-Web repository for scientific paper reviews might contain a partial copy of the submitted papers (e.g. the title and the names of the authors), some additional information added by the reviewers (e.g. the names of the reviewers and the reviews) and some mapping rules that allow to extract the abstract and the bibliography of the submitted paper. Then, the query "Give me the title of all papers that are reviewed by an author appearing in the bibliography" can only be answered by using all three kinds of information.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data On the Web: From Relations to Semistructured Data and XML.* Morgan kaufmann, October 1999.

[2] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, B. Amann, I. Fundulaki, M. Scholl, and A-M. Vercoustre. Managing RDF Metadata for Community Webs. In *Proceedings of the 2nd International Workshop on The World Wide Web and Conceptual Modelling*, pages 140–151, 2000.

[3] B. Amann, I. Fundulaki, and M.Scholl. Integrating ontologies and thesauri for RDF schema creation and metadata querying. *International Journal of Digital Libraries*, 3(3):221–236, October 2000.

[4] B.Amann and I.Fundulaki. Integrating Ontologies and Thesauri to Build RDF Schemas. In A-M. Vercoustre S. Abiteboul, editor, *Proc. of the 3rd European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, volume 1696 of *Lecture Notes in Computer Science*, pages 234–253, Paris, France, September 1999. Springer Verlag.

[5] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and L. Stefanescu. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery, February 2001.

[6] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Informal Proc. of ACM SIGMOD Workshop on Web and Databases (WebDB)*, pages 53–62, Dallas, USA, May 2000.

[7] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 141–152. ACM, 2000.

[8] J. Clark and S. DeRose (eds.). XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xpath.

[9] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5), 1998.

[10] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. To appear in Proceedings of the 27th International Conference on Very Large Databases (VLDB), September 2001.

[11] S. DeRose, E. Maler, and D. Orchard (eds.). XML Linking Language (XLink) Version 1.0. W3C Proposed Recommendation, December 2000. http://www.w3c.org/TR/xlink.

[12] J. Clark (ed.). XSL Transformation (XSLT) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xslt.

[13] Bancilhon Francois, Delobel Claude, and Kanellakis Paris. *Building an Object-Oriented Database System : The Story of* O_2. Morgan Kaufman, 1992.

[14] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An Information Integration System. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 539–542. ACM Press, 1997.

[15] F. Goasdoué, V. Lattés, and M-C. Rousset. The use of CARIN language and algorithms for information integration: The PICSEL System. *International Journal on Cooperative Information Systems*, 2000.

[16] P. Grosso and D. Veillard (eds.). XML Fragment Interchange. W3C Candidate Recommendation, February 2001. http://www.w3.org/TR/xml-fragment.html.

[17] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 299–310. ACM Press, 1999.

[18] A. Levy. Answering queries using views: a survey. http://www.cs.washington.edu/homes/alon/site/-files/view-survey.ps. submitted for publication.

[19] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[20] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995.

[21] Rachel Pottinger and Alon Y. Levy. A Scalable Algorithm for Answering Queries Using Views. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 484–495. Morgan Kaufmann, 2000.

[22] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/-pp/xql.html, 1998.

[23] Arnaud Sahuguet. KWEELT, the Making-of : Mistakes Made and Lessons Learned. Technical report, Department of Computer and Information Science, University of Pennsylvania, November 2000. http://db.cis.upenn.edu/DL/kweelt-TR.pdf.

[24] P. Wadler. A formal semantics of patterns in xslt, 1999. URL: citeseer.nj.nec.com/wadler99formal.html.