CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

Année : 2003 *No attribué par la bibliothèque*

THÈSE

presentée pour obtenir le grade de :

Docteur en Sciences

Discipline: Informatique

par

Irini FUNDULAKI

Titre de la thèse :

Intégration et Interrogation de Ressources XML pour Communautés Web

Date de Soutenance : 13 janvier 2002 devant le jury composé de :

Mr.	**M. SCHOLL**	**Directeur**
Mr.	**G. GARDARIN**	**Rapporteur**
Mr.	**N. SPYRATOS**	**Rapporteur**
Mr.	**B. AMANN**	**Examinateur**
Mr.	**C. BEERI**	**Examinateur**
Mme	**I. WATTIAU**	**Examinatrice**

Abstract

In this thesis we have studied the problem of *querying* and *integration* of *heterogeneous* and *autonomous* XML resources. Our contribution is two-fold : first we have examined the problem of the construction of *metadata schemas* by the *integration* of *ontologies* and *structured vocabularies (thesauri)*. Second, we have elaborated a model for the integration and querying of XML resources using the *mediator-wrapper* architecture where the schema of the mediator is an *ontology*.

In the first part of our work we developed a methodology for the construction of metadata schemas by the integration of an ontology and of structured vocabularies (thesauri). Ontologies describe the generic structures in the domain of interest by *concepts* and *roles*. Thesauri are vocabularies of terms with precise semantics which are not well-structured. Ontologies have a double role in our model : they define a generic view of information and a structural interface over thesauri. The resulting metadata schema allows the description of a large number of different resources using the generic schema provided by the ontology and the precise semantics of thesaurus terms. The results of this research were validated by the prototype ELIOT developed in the context of a contract between CNAM-Paris and the Service de l'Inventaire of the French Ministry of Culture.

In the second part of our work, we have studied the integration and querying of heterogeneous and autonomous XML resources. Our approach, ST_YX , is based on the *mediator-wrapper* architecture where the global schema of the mediator is an *ontology* which is not materialized : the actual data resides in the sources.

Our contributions in this context are multiple. First we have defined a simple but expressive model for describing XML resources. The resources are described by means of *mapping rules* between XML fragments specified by *XPath location paths* and *ontology paths*. The use of an ontology at the mediator level (which can be perceived as a conceptual schema with symmetric and inheritance relations) and the XPath language, allows one to represent a large number of XML resources. In addition, the approach of *path-to-path* mapping allows one to attribute specific semantics to the *parent/child* relationship between the nodes in an XML document.

We have developed a *rewriting* algorithm for *tree queries*, which transforms a user query formulated in terms of the ontology, into one or more XQuery queries expressed in terms of the local sources schema. User queries are tree queries with no joins, restructuring or aggregation. The rewriting algorithm calculates the *variable to rule bindings* by examining each query variable in the context of its father. The algorithm considers both the *full bindings* (*complete answers*) and the *partial bindings* (*partial answers*) between the query variables and the mapping rules. In this last case, the query is decomposed recursively in a prefix query which is evaluated by the source and one or more suffix queries which are considered for evaluation by the remaining sources. The *join* operation is used to complete the partial answers obtained by the evaluation of these queries.

In this context we have addressed the problem of identification of information. The XML resources which we consider are heterogeneous and autonomous. In consequence, the assumption of persistent object identifiers is not realistic. Towards this direction, we have introduced the notion of *global keys* at the ontology level : a key is a set of paths which identify the instances of a concept. Each source can in this way control the fragment identifiers exported by the mapping rules associated with the key paths. The ST_YX rewriting algorithm considers the presence of keys for the decomposition of the queries and the construction of the results.

Ithaca

When you set out on your journey to Ithaca,
pray that the road is long,
full of adventure, full of knowledge.
The Lestrygonians and the Cyclops,
the angry Poseidon – do not fear them:
You will never find such as these on your path,
if your thoughts remain lofty, if a fine
emotion touches your spirit and your body.
The Lestrygonians and the Cyclops,
the fierce Poseidon you will never encounter,
if you do not carry them within your soul,
if your soul does not set them up before you.

Pray that the road is long.
That the summer mornings are many, when,
with such pleasure, with such joy
you will enter ports seen for the first time;
stop at Phoenician markets,
and purchase fine merchandise,
mother-of-pearl and coral, amber and ebony,
and sensual perfumes of all kinds,
as many sensual perfumes as you can;
visit many Egyptian cities,
to learn and learn from scholars.

Always keep Ithaca in your mind.
To arrive there is your ultimate goal.
But do not hurry the voyage at all.
It is better to let it last for many years;
and to anchor at the island when you are old,
rich with all you have gained on the way,
not expecting that Ithaca will offer you riches.

Ithaca has given you the beautiful voyage.
Without her you would have never set out on the road.
She has nothing more to give you.

And if you find her poor, Ithaca has not deceived you.
Wise as you have become, with so much experience,
you must already have understood what Ithacas mean.

Constantine P. Cavafy (1911)

0

Contents

Chapter 1

Introduction

1.1 Web Communities

During the last decade, the Web has become the basic infrastructure for a large number of human communications and information-based activities. Web data comes from different organizations, communities and individuals, and covers a huge and diverse spectrum of interests.

Due to the wide use of the Web as the basic means for exchanging information, a number of *Web communities* have been created. Members of such communities are persons interested in a *specific domain* ranging from culture, education, music to business and have in common some *identified* and *formalized* knowledge that they use and make evolve in their joint activities along with a variety of *heterogeneous information sources* (e.g. documents, data). In this context, the main challenge is to provide a *single point of access* to the various information sources. Towards this direction *community web portals* have been developed whose basic objective is to provide to their users the means to *publish* and *access* in a meaningful way diverse information resources in the domain of interest.

Recently, the *Semantic Web* initiative was launched to handle the issues related to the deployment of *Web community portals*. The goal of the Semantic Web is to *"develop enabling technologies and standards to support richer discovery, data integration, navigation and automation of tasks"* [221]. The idea behind this initiative is to build on top of the actual data, a layer that actually describes the *semantics* of Web data, independently of the format in which it is represented.

One of the basic issues in a Web community portal is that information sources are developed in an *autonomous* way. This autonomicity results in sources which are *highly heterogeneous* in *structure* and *access interfaces*. Data can be *structured* coming from *relational* [147] and *object oriented* [123] databases, *semi-structured* [3, 31] (e.g. XML data [4]) and finally *unstructured* documents (e.g. HTML documents or documents in proprietary formats such as Postscript, PDF etc.). As far as *access interfaces* are concerned, a relational source supports *SQL* [192] queries and an object oriented one *OQL* queries [53, 42]. HTML documents in a Web portal can be accessed by simple keyword-based interfaces similar to those provided by search engines such as Google [99], Altavista [14] and Yahoo [223] or by more elaborate interfaces using HTML forms.

An orthogonal problem to that of heterogeneity concerns the distribution of data. Due to the autonomous way in which information sources are developed, the information a user is looking for is not located in a single source but can be found in several sources. For example, in a cultural portal, one source might contain descriptions of Van Gogh paintings, and a second one studies of the paintings. Consequently, a user who looks for Van Gogh's paintings and studies about them must access both sources (using the source specific interfaces). During these searches, a user might

possibly access *several irrelevant* sources to discover the information she looks for. Hence, due to source heterogeneity and distribution of information, looking for information in a Web portal is a cumbersome task.

To support efficient access of information sources, a Web community portal can be considered as a *database* [210] providing users with a *single point of access* (i.e. *schema*) and a *single query interface*. The *portal schema* is the *backbone* of the portal which provides to the users a single point of access to the sources and describes the basic *notions* in the domain of interest. To deal with the autonomicity and heterogeneity of the sources, it is important that portal schemas be built *independently* of the structures of the underlying sources. They can either be constructed from scratch, or by reusing *existing conceptual* structures such as *ontologies* [103] and *terminological structures* such as *thesauri* [114].

Although the development of XML [4] is slowly changing the picture for the exchange and representation of Web data, a large number of resources in a Web portal are basically HTML documents either produced dynamically from databases or created by community members. One way to make the sources, which do not have a well defined structure, accessible in a community web portal, is to provide appropriate *metadata* [190]. Metadata can be distinguished between *content descriptive* metadata which capture the semantics of a source that is not explicitly present in its content, *structure descriptive* which are related to the source structure and finally *administrative metadata* which concern information such as the document's date of creation, access rights etc.. Sources are basically *published* in a portal by providing their *content descriptive metadata* defined in terms of the portal schema. Metadata description languages like *RDF* [179], XML Topic Maps [166] and more recently DAML+OIL [59] were developed for this purpose.

Users can formulate *structured queries* in terms of the portal schema and the portal is then responsible for exploiting the available content descriptive metadata to *discover* those that contain information relevant to the user's request. *Resource discovery* [159] is one of the basic functionalities that a portal offers to its users. The result of a user request is a set of *source addresses* which contain information relevant to the user query. Then the user can navigate in each of the sources to locate the information she looks for.

Although metadata is one way of answering user requests, users need to *actually query the contents of information sources*. Consider again a cultural portal where sources publish information about cultural artifacts, expositions in museums and reviews for them. Information about an artifact can be found from several sources. A user looking for *"the French museums with Van Gogh expositions and reviews"* must find the sites of French museums, search for expositions of Van Gogh paintings and then for these expositions discover reviews from specialized sites. In this case, the user must 'manually' combine information found from a possibly large number of information sources. The objective here is to relieve the user from this cumbersome task : in other words, the functionality that should be provided by a Web portal is that of a *data integration system*.

A number of research efforts during the last years focused on the problem of data integration. The first generation of such systems are *multidatabases* [142] and *federated* [110] systems. Nevertheless, given that we are in the Web environment the fact that sources are *autonomous* must be taken into account. Due to the preservation of autonomicity, rigid data integration architectures, based on the integration of local sources schemas introduced by multidatabase systems [142], are difficult to apply. Wiederhold in [216] proposed the *mediator-wrapper* architecture which has been extensively used in data integration systems. The backbone of the mediator is a *mediated* or *global* schema which offers a single view of the underlying data. The global schema is not materialized : data resides in the actual sources. To access this data, sources are integrated in the mediator by providing their *source descriptions*. User queries are formulated in terms of the global schema and are rewritten to

queries expressed in terms of the local sources schemas using the source descriptions.

Before the emergence of XML [4] a large number of data integration projects concerned the integration of *structured* [136, 200, 47, 97, 154, 202] and *semi-structured* [94, 54] sources. The emergence of XML has changed the picture. A number of research projects have proposed XML as a language to express a common interface between existing databases or to express a uniform view to integrate heterogeneous data sources [149, 187, 70, 87, 88]. This thesis is concerned with the *querying* and *integration* of XML resources.

1.2 Contributions

In this context we propose a *framework* for the deployment of *Web communities portals* supporting the following functionalities :

- the creation of *portal schemas* that define the basic notions in the domain of interest which can be used as *metadata schemas* and as *mediator schemas* [18];
- the *publication* of Web resources in terms of their *content descriptive metadata* [19, 12] and
- finally the *querying* and *integration* of *autonomous* and *heterogeneous* XML resources [17, 16, 15].

This framework has been validated in the context of various projects in the cultural domain [62, 157] and different prototype implementations [176, 92].

1.2.1 Creation of Portal Schemas and Content Descriptive Metadata

In a number of specific applications or domains of interest, there already exist specific *ontologies* [146] that describe the semantics of the domain. For example, in the cultural domain which is the reference example in this thesis, there exists the ICOM/CIDOC Reference Model [75] which is the result of an effort undertaken by the International Council of Museum Documentation [113]. This model is used in a consistent way for the exchange of data coming from different museums. In addition to these ontologies, a large number of controlled vocabularies or *thesauri* have also been developed and extensively used. For example, for the cultural domain there exist (among others) the *Art & Architecture Thesaurus* [194, 2] and the ULAN [203] thesaurus from the Getty Institute, the MERIMEE [156] thesaurus of the French Ministry of Culture and the RCHME [178] thesaurus of the Royal Commission of Historical Monuments of England. Thesauri contain several hundreds or thousands of terms to describe the concepts in a specific domain and are used as efficient means for *consistent indexing* and *retrieval* of information [90].

Our idea was to produce *rich portal schemas* by the *integration* of *ontologies* and *thesauri* [18] which can be considered as orthogonal ways for describing information. The former provide structural, sharable views of information, with usually shallow semantics. They are declarative specifications of the *concepts* and *roles* in a domain of discourse. Thesauri are structured vocabularies, with rich semantics but little or no structure. For example, although the *Art & Architecture Thesaurus* includes extended taxonomies of cultural artifacts and styles, there is no explicit relationship expressing that artifacts are associated with a style. In the context of our approach, ontologies have a dual role : provide a *generic view* of information and a *structural interface* over thesauri.

As mentioned before, portal schemas can be used as *metadata schemas* for the controlled generation of *content specific metadata*. In this thesis we show how ontologies enriched with thesauri are

very useful for the creation of such metadata and we propose a *resource description language* [19] for this task.

We have implemented a research prototype to validate our approach for the development of rich metadata schemas [176] where we followed an object-oriented approach using the OODBMS O_2 [91] for storing the resulting schema, and the source metadata. These descriptions are then queried using OQL [53], the standard language for querying object-oriented databases.

Our approach has been validated by the European research project C-Web (Community Webs) [50, 62] and by a national contract between the French Ministry of Culture [66] and the Conservatoire National des Arts et Métiers in Paris [209].

The purpose of the C-Web project was to support specific communities (e.g. in commerce, education, culture, health sectors) that share formalized knowledge in form of an *ontology* and information sources (like documents or data sources). In order to support Web communities, C-Web offers a number of functionalities : (i) the *creation of conceptual schemas*, (ii) the *publication* of metadata for community resources and (iii) the *querying* of information sources using the available metadata.

The basic objective of the contract between the CNAM and the French Ministry of culture was to create a portal for the different services of the ministry. In a first step we collaborated with people from the different services to define an *ontology* that would capture the basic notions in the domain of discourse. The second task was to create a rich portal schema by integrating the ontology with the available thesauri following our approach. Last, the resulting schema was used to record content descriptive metadata for the digital documents of two services within the divisions of *"Patrimoine et Archéologie"*. The result of this collaboration was the ELIOT cultural portal [176].

1.2.2 ST_YX : Ontology-based Integration of XML Web resources

In the context of the Semantic Web, we propose a new approach for the integration of *heterogeneous* and *autonomous* XML resources in a Web community, based on the *mediator-wrapper* architecture and following the *local as view* approach which is validated by the ST_YX prototype [92]. To summarize, the contributions of this thesis in the area of XML data integration are :

Ontology-based Mediation : Most of the integration systems dealing with the integration of XML data which follow the local as view approach use either a *relational schema* or an XML DTD as the mediator schema. In contrast, we use an *ontology-based* mediator [15] and we shall argue why this approach better responds to the challenges set by the Semantic Web. The ontology is defined *independently* of the actual source structures and contains the basic notions in the domain of interest. *Concepts* in the ontology are used to represent *entities*, *symmetric binary roles* between *concepts* describe *semantic relationships* between *entities* and finally *attributes* represent the entities' *properties*. Finally, concepts can be related by *inheritance* relationships which, as in traditional object oriented models, represent commonality of structures.

Mapping Language : *Mapping rules* that associate XPath [52] *location paths* to *ontology paths* [17, 16] are used to publish XML resources in ST_YX. Mapping rules allow one to describe an XML resource as a *view* of the global schema, as done in the *local as view* approach, independently of the other sources. The major benefit of this approach is that any modification of a source does not affect neither the mediator, nor the descriptions of the other sources.

The use of ontology paths in the mapping rules, allows one to describe source structures which cannot be directly mapped to the ontology. This need arises from the nature of the ontology which is

defined *independently* of the structure of the sources and contains concepts which are not necessarily represented in a source.

The use of XPath location paths to describe the XML source structure was dictated by the following reason : when this work started, there existed a limited number of XML resources available on the Web. These resources were basically XML documents stored in some Web server and the only way to access them was by their URL. Hence, these sources were not able to evaluate queries expressed in some full fledged XML query language. But, existing software can be installed very easily on top of Web servers and used to evaluate simple XPath expressions.
Moreover, XPath *is the standard* for addressing XML data and is used in a number of XML query languages (XQuery [44], XQL [71], Quilt [45] among others) and XML related technologies (XSLT [82], XPointer [186] etc.). More specifically, XPath is extensively used by XQuery which, hopefully, will be the standard XML query language. Because of its expressive power and its simplicity w.r.t. more complex languages, XPath as not only as a language for adressing XML document parts but also as a query language, is expected to last for a significant time.

Query Evaluation : The user queries the information sources using *simple tree queries* formulated in terms of the ontology.

To obtain answers to a query, this must be evaluated against all sources that might contain all or part of the information requested. To evaluate a query over an XML resource, this must be *rewritten* into an XML query (e.g. XQuery) expressed in terms of the local source's schema. During this rewriting, the mapping rules that provide answers to the query are selected. It might be the case, that when attempting the rewriting of a query for a source, full answers from the source are not obtained (i.e. the source does not give answers to all the query variables). In this case, in contrast to [55] we attempt to complete these partial answers by *decomposing* the query. The result of this decomposition is (i) the query that the source can answer and (ii) a set of subqueries that can be possibly answered by the other sources. The partial answers obtained from the sources are then *joined* at the mediator site.

Using the mapping rules selected by the rewriting process, the user query is rewritten into an XML query expressed in terms of the local source's schema.

To perform the joins between the partial results we need a way to identify the XML fragments obtained by the sources. Another contribution of our work consists in the definition of *keys*. We introduce two different types of keys : *local keys* which are used to decide whether two XML fragments correspond to the same XML fragment and *global* or *semantic keys* which are used to identify XML fragments that originate from different or the same source. Most of the data integration projects use either persistent object identifiers [149], or identifiers created using Skolem functions [20, 170, 54]. Both approaches are based on the assumption that information sources have some common way to identify their objects and do not consider the case where this is not true. In contrast, our approach on introducing global keys at the ontology level which are defined independently of the sources supports a common way of object identification.

1.3 Organisation of the document

This thesis is organized as follows :

Chapter 2 is a state of the art on *ontologies* as well as *web data integration* issues. First, the notion of *ontologies* as encountered in computer science and more specifically in the field of knowledge engineering and issues related to ontologies such as *ontology representation languages*, *classification* and *examples* of ontologies are presented. Second, *metadata related issues* are discussed such as the

state of the art metadata representation languages : RDF [133], DAML+OIL [59] and XML Topic Maps [166]. This part of the chapter is concluded with a discussion on *HTML document annotation systems* which can be considered as one of the first approaches for specifying the semantics of HTML documents by means of *annotations* which can be considered as a kind of *content descriptive metadata*.
In the second part of the chapter issues related to Web data integration are presented. We first discuss informally the *local* and *global* as view approaches and then describe a number of systems that follow these approaches by examining their (i) *data model*, (ii) *query language*, (iii) *source description language* and (iv) *query rewriting algorithms*.

Chapter 3 describes our contribution related to the *construction of portal schemas by integration of existing ontologies and thesauri*. The notions of *ontologies* and *thesauri* are formally presented and then we illustrate in detail how a portal schema can be built out of these structures. The resulting portal schemas are language independent and can be represented in different ways. We first illustrate a simple translation to RDF schemas [179], a standard language for the definition of metadata schemas. Then we discuss different translations into an object oriented model and their efficiency w.r.t. query processing. The chapter is concluded by the presentation of the ELIOT [176] Web portal developed in the context of a National contract between the French Ministry of Culture and the Conservatoire National des Arts et Métiers.

Chapter 4 discusses our contribution for the querying and integration of heterogeneous XML resources using an *ontology-based* mediator. We first define the ST_YX mediator model and *mapping language*. Then we present (i) the ST_YX query language, (ii) the query rewriting and query decomposition algorithms and (iii) the generation of query execution plans. The chapter is concluded by a discussion on the choices for the ST_YX global schema and mapping language and by a presentation of the ST_YX prototype which implements our approach.

Finally Chapter 5 summarizes the work of this thesis and presents some interesting research directions for future work.

Chapter 2

Semantic Data Integration

During the last decade, the Web has become the basic infrastructure for a large number of human communications and information-based activities. Web data comes from different organizations, communities and individuals, and covers a huge and diverse spectrum of interests. The principle of the Web is that information is exchanged in the form of files, that are uniquely identified by a *Uniform Resource Locator (URL)*. Web data is mostly encoded using the HTML (Hypertext Markup Language), which is the lingua franca for publishing hypertext information on the Web.

Users issue *keyword-based* queries to *search engines* such as Google [99], Altavista [14] and Yahoo [223] to discover pages of interest. The results of these searches are lists of URLs of Web pages which contain the set or a subset of the query keywords. From this set of pages the user must look at *each* of them to find the page that contains the information she looks for.

A first attempt to better organize Web data in order to access it in more meaningful ways, is the recording of *descriptive information* in the form of *metadata* [190]. Metadata can describe either the *content* or *administrative* information about actual data. *Metadata vocabularies* have been used for (i) the *creation* of metadata for digital information sources and (ii) *resource discovery*. Whereas metadata based systems can be used for *resource discovery* they do not support the *querying* of the source contents. This is the purpose of *data integration systems* which allow their users to query *autonomous* and *heterogeneous* information sources like a *single source* with a *unique schema* and *query interface*.

The common component in resource discovery and data integration systems is the use of a *single schema* of the underlying sources. In the former this is provided by a *metadata vocabulary* and in the latter by a *mediated* or *global schema*. *Ontologies*, which capture the basic notions of a domain of interest and are developed *independently* of the source structures by domain specialists and knowledge engineers, can be used for both purposes.

In this chapter issues related to *ontologies and thesauri* are presented (Section 2.1). More specifically, we discuss the *notion* of ontology and how an ontology can be *used* to resolve the different kinds of *heterogeneities* encountered in digital information sources. We also give Guarino's classification of ontologies and describe the notion of *thesauri* (Section 2.1.2). Examples of ontologies are also given and the most important ontology representation languages are discussed. Metadata related issues are presented in Section 2.2 including some representative metadata languages (RDF [30, 133], DAML+OIL [59] and XML Topic Maps [166]).

Web data integration systems are discussed in Section 2.3. First we present the general idea behind data integration systems, and show how the dimension of the Web affects their architecture. Then, the two basic approaches for describing the source contents to the system are illustrated : the *local as view* and the *global as view* approaches. The most important systems that follow these

approaches are presented in Sections 2.4 and 2.5. For each of the systems we examine (i) its *data model*, (ii) the language used for the *source descriptions* and (iii) the *query rewriting algorithms*.

2.1 Ontologies and Thesauri

2.1.1 Ontologies

The term *ontology* was first introduced as a *"philosophical discipline, a branch of philosophy that deals with the nature and the organization of reality"*. It was presented by Aristotle in his work about Metaphysics as the *science of being* which tries to answer the following questions such as *"what is being"* and *"what are the features common to all beings?"*.
In the computer science field, the term ontology refers to an *engineering artifact* which :

- is constituted by a specific vocabulary used to describe a certain reality (i.e. *domain of interest*);
- and a set of explicit assumptions regarding the intended meaning of the vocabulary.

Ontologies are formal specifications of a specific domain and provide a shared understanding of the domain. According to Gruber [101] an ontology is *"an explicit specification of a conceptualization"* whereas Uschold and Gruninger [206] define an ontology as a *"shared understanding of some domain of interest"*.

In the context of our work, an ontology captures the basic notions in a domain of interest : *entities*, *semantic relationships* between entities and *properties* of entities. The term *concept* is used to denote the entity types in the domain of discourse. The New Oxford Dictionary of English gives the following definition for the term *concept* :

"An idea or thought that corresponds to some distinct entity or class of entities, or to its essential features, or determines the application of a term, and thus plays a part in the use of reason or language."

The notion of concept as it is presented in Description Logic languages, accounts to a *set* or a *class* of individual objects [165]. In Artificial Intelligence the notion of *frame* and in Object Oriented paradigm the notion of *class* are similar to that of a concept.

A tremendous amount of work has been done recently concerning the use of *ontologies* as the basic means to resolve the *semantic* heterogeneities between data originating from different sources. Semantic heterogeneities fall into the broader category of *logical* heterogeneities [119]. This term is used to denote the *semantic*, *schematic* and last *structural* heterogeneities between the sources.

Semantic Heterogeneities concern the semantics of data and schema. Semantics of data is defined by Wood in [219] as the *"the meaning and the use of data"*. For example, a database schema consists of constructs (e.g. relations/classes, attributes) which have a *name* denoting the real world concepts and relationships. For example, for the same concept, different applications might use different names : they may both talk about persons, but one refers to this concept using the name "Person" while the other uses the name "Human". This is the case of *synonyms*. On the other hand, the same name can refer to two different concepts (*homonyms*).

Schematic Heterogeneities refer to the different encoding of the *same concept* in different applications or database schemas. For example, in a relational source, a concept can be recorded

as a relation, an attribute, or an attribute value. In the XML context, a concept can be represented as an element an attribute, or an attribute value.

Structural Heterogeneities arise when the same concept is represented in the same data model by different structures. For example, two relational applications might model the concept of person using the relation Person but associating different attributes with it.

The above types of heterogeneities lead to poor communication between users and systems. People, organizations and software systems associated with a *specific domain of interest* communicate between them using an *ontology* which has *well established* semantics, is defined *independently* of the underlying structures and is *agreed by the members of the community.*

In some broader sense, ontologies (of all kinds) can be used not only for describing the exact structure of information sources in order to resolve the semantic heterogeneities between them, but also to represent *metadata.* In the Web context, *ontologies* are used to exchange *metadata* between digital information resources. Ontologies in this case are called *metadata vocabularies.* Metadata related issues are presented in Section 2.2.

There are a number of different ways of *describing, using* and *organizing* ontologies. We present here the classification proposed by Guarino in [104] concerning the *use* of ontologies :

- *Top-level ontologies* describe very general concepts such as space, time, event which are independent of a particular problem or domain. If we are concerned with large communities of users interested in a variety of domains it is necessary to have such wide-spread ontologies.
- *Domain ontologies* are specializations of top-level ontologies. They describe a vocabulary related to a domain of interest by specializing the concepts introduced by the top-level ontology.
- *Task ontologies* are specializations of top-level ontologies. They describe the vocabulary related to a generic task or activity by specializing the top-level ontology concepts.
- *Application ontologies* are the most specific ones. The concepts introduced here correspond to entities associated with a specific domain of interest and with a specific activity.

Examples of top-level ontologies are Wordnet [161, 220], the Upper CYC Ontology [63] and the IEEE Standard Upper Ontology [205]. WordNet falls into a specific class of ontologies, called *lexical semantic networks.* Its scope goes beyond the scope of an application or a task and covers an extensive set of lexico-semantic categories or *synsets* as they are usually referred to. A word in Wordnet is associated with a *meaning*, and belongs to a *synset.* A *synset* organizes words that have similar meanings. Words belonging to a synset are also characterized with a *syntactical* form : adjective, adverb, verb and so on.

Synsets are organized using predefined semantic relationships : *synonymy*, *antonymy*, *hyponymy*, *meronymy* [162]. According to Oxford's dictionary, *"two words are synonymous, if they have the same meaning in the same language, though perhaps with a different style, grammar or technical use".* For example the words 'kill' and 'murder' are synonymous. On the other hand, *"two words are antonyms if they have opposite meanings"* : words 'old' and 'new' are antonyms. Hyponymy is a relationship that organizes synsets into hierarchies. The higher a synset is found in a hierarchy, the more general is the meaning of the words that belong to the synset. Finally, *meronymy* is the relationship used to represent *part-whole* relationships (e.g. the 'cardio-vascular system' is part of the 'circulatory system').

Another example of a top level ontology is the Upper CYC ontology which is a formalized representation of a *'vast quantity of fundamental human knowledge"* and contains about 10^5 general concepts and 10^6 assertions on these concepts. Its scope is not restricted to any domain or application and is used to represent knowledge in a large number of domains.

Examples of domain and application-specific ontologies are the UMLS [204], Dublin Core [79] and the KA2 Ontology [116]. The UMLS (Unified Modeling Language System) contains a number of knowledge sources : (i) a metathesaurus which contains 730.000 concepts of the medical domain (ii) a lexicon which provides synonyms, lexical variants and grammatical forms of words appearing in the meta-thesaurus and (iii) a semantic network that encodes the relationships such as *cause*, *symptom* between medical terms.
Dublin Core is a simple ontology for recording administrative data about digital documents and is presented in Section 2.2. Finally, the KA2 ontology models the basic concepts in the *knowledge acquisition domain* (like researchers, topics, products etc.).

2.1.2 Thesauri

Thesauri contain several hundreds or thousands of terms often associated with a specific *domain of interest* and can be classified as *application ontologies*. They are used as efficient means for *consistent indexing* and *retrieval* of information [90] and as data value standards from the point of documentation or cataloging. In this context they can be considered as a controlled vocabulary or an authority [74]. They can also be used as assisting tools in database retrieval systems [169].

A thesaurus according to [114] is a *"structured set of controlled terms. Terms are selected from natural language and used to represent, in summary form, the subjects of documents"*. A term is *controlled* if it is considered by the experts in the domain as an *unambiguous term* to name a *real world concept*. Terms in the thesaurus are organized using a *priori* defined relationships. Examples of such relationships are *broader/narrower* and *related* term relationships.

A tremendous amount of work has been done for the development of thesauri in a number of different domains. For example, for the cultural domain there exist (among others) the *Art & Architecture* [194, 2] and the ULAN [203] thesauri from the Getty Institute, the MERIMEE [156] thesaurus of the French Ministry of Culture and the RCHME [178] thesaurus of the Royal Commission of Historical Monuments of England. The scope of the first one is art, architecture, decorative arts, material culture and archival materials. It contains 120.000 terms and its coverage ranges from antiquity to the present. The second contains an extended list of artist names (220.000 names), or corporate bodies (i.e. groups of people working together to create an artifact).

Thesaurus Structure Thesaurus terms are considered as the *"representation of concepts in the form of a noun or a noun phrase"* [114]. Concepts are perceived by thesaurus developers as referring collectively to a set of objects (*concept instances*) [158] that are considered as such not with respect to a formal classification process but through a common agreement. Under this perspective, the interpretation of a thesaurus term is a set of concept instances, which is called the *extension* of the term. Thesauri include a fixed set of semantic, *a priori* defined, term relationships. Due to the set theoretic definition of terms, a subset of these relationships can be interpreted as relations between sets [115, 193, 76]. A thesaurus allows terms that are associated with a subject to be regrouped into *hierarchies* which are cross-referenced with other groups of terms that can be relevant to this subject.

Thesauri can be monolingual or multilingual. In the first case, terms originate from a single language and in the second, terms originate from different languages. In this work we consider *only*

monolingual thesauri defined according to the ISO 2788 standard [114] for the documentation and establishment of such thesauri.

The ISO 2788 standard distinguishes between *preferred* and *non-preferred* terms. A preferred term is used *unambiguously* to represent a given concept. This term is also referred to as a *descriptor* and is considered to be the lexical identifier of the concept. From this point on, and for thesauri we use interchangeably the words descriptor and concept. A thesaurus might contain descriptors denoting *concrete concepts* ("paintings", "sculptures", "clay"), *abstract concepts* ("impressionism", "oil on canvas"), *disciplines or sciences* ("archeology", "mathematics"), *units of measurement* and finally individual entities ("Holland", "Vincent Van Gogh").

Non-preferred terms are *synonyms* (or quasi-synonyms) of descriptors. Although such a term can be assigned to a document, it cannot be used as an entry point neither to a thesaurus nor to an alphabetical index.

In the majority of thesauri, each descriptor is identified by a record. A record is associated with a *unique ID*. For each descriptor, there is a note in natural language explaining its scope. It is also associated with a list of non-preferred terms (*synonyms*) that can be used as its alternatives when querying. A descriptor can also have *related descriptors* which have same or overlapping meanings.

Relationships between thesaurus terms are represented by the ISO 2788 standard *hierarchical*, *equivalence* and *associative* intrathesaurus relationships.

Hierarchical relationships : This is the basic type of thesaurus relationships which distinguishes a systematic thesaurus from an unstructured list of terms (such as a glossary or a vocabulary). Hierarchical relationships include the *generic*, *whole-part* and *instance* relationships.

The generic relationship (called *broader term generic*), denoted by *btg*, is the *generalization* relationship between concepts. For example this relationship records the fact that *"painting"* is a *"work of art"*. The inverse of the *btg* relationship is *narrower-term generic*, denoted by *ntg*. The majority of thesauri are constructed using the broader-term generic *btg* relationship.

The whole-part relationship (called *broader-term partitive*) is denoted by *btp*. This relationship designates that a concept is part of another. It covers a limited range of situations where the name of a part implies the name of its whole in some context. This relationship allows to represent facts such as for example that the *"cardio-vascular system"* is part of the *"circulatory system"*. Last, the instance relationship (called *broader instance*) is denoted by *bt*. It identifies a link between a general category of things or events expressed by a common noun and an individual instance of this category. For example, this relationship is used to describe the fact that *"Vincent Van Gogh"* is a *"painter"*.

Relationships of this kind are only used to connect descriptors. Since each descriptor is the lexical counterpart of a concept, and the latter is associated with instances, the above relationships have well defined semantics.

Equivalence Relationship : The *use* relationship is used to associate a descriptor to one or more non-preferred terms. Its inverse is the *uf* (*used-for* relationship). Both relationships do not have well defined semantics. A non-preferred term does not denote a concept but is seen as a lexical alternative of the descriptor denoting the concept, hence it has no set-theoretic interpretation.

Associative Relationship : It is used to record relationships between descriptors that can neither be considered as equivalent, nor be associated by one of the previously mentioned hierarchical relationships. This relationship, called *related-term* relationship, is denoted by *rt* and is reflexive. Usually the descriptors associated with such a relationship, denote concepts that have

overlapping meanings. Related descriptors are associated to such an extent that the explicit representation of their relation would reveal alternative descriptors that are used for indexing. In information retrieval, this relationship is used to broaden the result set of a user query. It does not have well defined semantics but it is very common in monolingual thesauri. More specifically, it is used to denote associations between descriptors belonging to a) the same category (i.e. concepts with overlapping meanings, such as *boats* and *ships*) or b) different categories (i.e. terms referring to different conceptual types), such as *computer systems* and *data processing*).

2.1.3 Ontology Representation Languages

There are a number of proposals for ontology representation languages. According to Gruber [101] knowledge in ontologies can be formalized using five different kinds of components: *concepts*, *relations*, *functions*, *axioms* and *instances*. Nevertheless, not all languages support all five constructs.

Description Logics Description Logic languages [27, 77] is a family of languages suitable to represent a large class of ontologies. They distinguish between *class* and *instance* definitions. They support the definition of *concepts* and *roles* which stand for concept attributes and properties. Description Logic languages provide a set of *constructors* (such as *existential/universal quantification*, *conjunction* and *disjunction* of concepts, *number restrictions* on roles, and concept *inclusion*) to define more complex concepts (known as *concept expressions*). These constructors allow the definition of *taxonomies* of concepts and roles. The power of Description Logic languages is the possibility to define assertions for concepts and roles, as well as assertions for instances. They provide mechanisms to reason on whether a concept or role expression is *coherent*, *subsumes*, *is disjoint from*, or *equivalent* to another concept or role expression [21].

OIL OIL [85] is based on standard frame-based languages and its formalization is done through Description Logic style logical formulas. It allows the definition of *classes*, *properties* of classes and class *hierarchies*. For example, consider the definition for class Person appearing below.

```
1. class-def Person
2.         subclass-of Actor
3.         slot-constraint has_name
4.                         has-value String
5.                         max-cardinality 1
6.         slot born
7.                         has-value Event or Date
```

Class Person is defined as a *subclass* of class Actor (an actor can be a person, an organization etc.) (line 2). It has two properties : *has_name* and *born* (lines 3 and 6). The former takes its values in *type* String (line 4). It is also associated with the cardinality constraint *max-cardinality* (line 5) which states that a person can have at most *one* name. The latter takes its values in either class Event or class Date (line 7). OIL allows also to define that a class is disjoint or is not a subclass of another. For example, the following definition states that a person is not a subclass of Man_Made_Object.

```
class-def Person
        subclass-of not Man_Made_Object
```

F-Logic F-Logic [122] is a language that inherits notions from Object Oriented languages, Horn Logic and Deductive Databases. In F-Logic one can define *objects* and *methods*. Objects are referred to by unambiguous names, and have an identity that is independent of their values (as is the case in object oriented models). Methods allow the definition of relations between objects.

F-Logic *statements* express facts about objects (instances). For example, one can state that object 'Van Gogh' created objects 'Starry Night' and 'Vincent's room at Arles' by the following statement :

```
Van Gogh[created ->> { Starry Night, Vincent's room at Arles }]
```

F-Logic allows one to *define classes*, and *relationships* between classes. For example one could define that the relationship `created` is defined between the classes `Person` and `Man_Made_Object` by the following expression :

```
Person[created => Man_Made_Object]
```

It is possible to express in F-Logic that a class is a *subclass* of another. The expression `Person::Actor` states that class `Person` is a subclass of `Actor`.

F-Logic also supports the *declaration* of instances of classes : for example one can write that 'Van Gogh' is a person, or that 'Starry Night' is a man made object by the following expressions :

```
'Van Gogh'::Person, 'Starry Night'::Man_Made_Object
```

Finally, F-Logic supports the specification of rules which allow one to derive facts (assertions) from existing facts. For example, one could express the fact that if a person x is a father of a person y, then x is an ancestor of y :

$$FORALL\ x,y\ x[ancestor- >> y]\ \leftarrow\ x[father- > y]$$

One can query an F-Logic knowledge base by *queries* which are rules with an empty head. For example, if someone looks for *"the paintings created by Van Gogh"*, one can write the expression :

$$FORALL\ y\ \leftarrow\ Van\ Gogh[created- >> y :: Painting]$$

Notice here that variable y is restricted to be an instance of class `Painting` and not an object in general.

The advantages of F-Logic is that it has an expressive mechanism for reasoning with instances. Its object oriented view allows a good interaction with object oriented and relational systems.

2.2 Metadata

In the context of the Web, digital information sources are developed in an autonomous way. This autonomicity results in information sources which are highly heterogeneous in *structure*, *interfaces* and *semantics*.

To achieve interoperability between information sources, the approach of *metadata* has been introduced. Metadata are classified according to [189] into :

- *content independent metadata* which capture information such as the author, creation/modification date of the document (administrative metadata), the technical access mechanisms (communication protocols, source query capabilities etc.), the quality of the source (such as the reliability, update-frequency etc.) and user profiles.

- *content dependent metadata* which represent the content of information sources. These metadata are classified into :

  1. *logical metadata* such as the data dictionaries of relational database management systems, data guides of semi-structured data sources, or class diagrams in object-oriented database systems.
  2. *semantic metadata* : They describe the *semantics* of data, independent of any schema. This kind of metadata does not consider the data model of the source. *Ontologies* can be used to record semantic metadata about digital sources.

Metadata can support a number of functions such as information location and discovery and documentation of data. A great amount of effort has been invested in the development of metadata vocabularies for the exchange of information across different applications and domains.
The Dublin Core [212] ontology is a metadata vocabulary used to record administrative and semantic metadata about digital information resources. It consists of a common set of *elements* which can be used to describe in a consistent manner administrative and semantic metadata. Examples of elements in the Dublin Core element set are *title*, *creator*, *publisher* and *subject*. These elements are used to describe basically the administrative data about a document. Only the *subject* element is used to describe information about its actual content. Nevertheless, this element by itself cannot be considered as a sufficient way to model the semantic metadata of a document.
Another metadata vocabulary is the Warwick framework [130] which proposes a container record architecture comprising more and different types of metadata elements than those proposed by the Dublin Core. For example, elements specifying the terms and the conditions of the document's use, security information, authenticity, signatures. Another example of a domain independent metadata vocabulary is RSS (RDF Site Summary) [185] which is a lightweight multipurpose extensible metadata description and syndication format.

The above vocabularies are general purpose and their focus is mostly on describing the document's administrative data. Another example of domain or application dependent metadata vocabularies is USMARC [207] which is issued by the Library of Congress and defines a set of descriptive elements for the exchange of bibliographic items. In the cultural domain, the Aquarelle project [159] uses the SGML CI DTD of the French Ministry of Culture to describe a set of element names, dedicated to territory inventory making. Last the RETSINA [181] ontology provides interoperability between RDF based calendar descriptions on the Web and can be used in Personal Information Management Systems such as Microsoft Outlook.

2.2.1 Metadata languages

During the last years there has been number of formalisms developed for representing metadata vocabularies. Ontology languages can also be used to represent metadata schemas. The former come basically from Knowledge Representation and more specifically Description Logic languages. Metadata languages share several features with ontology representation languages but they are used for the representation and exchange of *Web* data.

PICS One of the first standards for describing metadata for Web resources is PICS (Platform for Internet Content Selection) [173] which consists of a set of specifications which enable people to distribute metadata about the content of digital material in the form of *labels*. It was originally

```
1.  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
2.   <rdf:Description about=
            "http://metalab.unc.edu/louvre/paint/monet/first/impression">
3.     <title>Web Museum: Monet, Claude: Impression: soleil levant</title>
4.     <presents>
5.       <rdf:Description about="soleil_levant">
6.         <type>oil painting</type>
7.         <title>Impression : soleil levant<title>
8.         <style>impressionism<style>
9.         <period>first-impressionism<period>
10.      </rdf:Description>
11.    </presents>
12.    <creator>Nicolas Pioch<creator>
13.  </rdf:Description>
14. </rdf:RDF>
```

Figure 2.1: An RDF description for resource `http://metalab.unc.edu/louvre/paint/monet/-first/impression.`

introduced to screen out materials unsuitable for children on the Internet. PICS led to the development of the Resource Description Framework (RDF) [179, 29] which is a *foundation for processing metadata.*

RDF Syntax and RDF Schema RDF can be used to describe any kind of *resource* [133] that is identified by a URI (Uniform Resource Identifier), such as a Web server, an XML document or an element of an HTML page (e.g. an image). RDF supports the definition of resource *properties* whose values can be other resources or literals (strings, integers). A collection of *property/value* pairs that refers to a specific resource is called an RDF *description* and can be represented as a labeled directed graph where nodes correspond to resources or literals (values) and edges to resource properties. RDF descriptions can be defined *independently* of any RDF schema.
Consider for example the RDF description about a painting of the French painter Claude Monet shown in Figure 2.1. Line 2 tells us that the description that follows concerns the HTML page which can be accessed by the URL `http://metalab.unc.edu/louvre/paint/monet/first/impression/`. The title of this page is *"Web Museum: Monet, Claude : Impression : soleil levant "* (line 3) and has been created by Nicolas Pioch (line 12). To describe properties of the painting, it is necessary to define a local resource which is identified by URI `soleil_levant` that refers to the painting (line 5). The painting's properties are its *type* (`oil painting`, line 6), *title* (`Impression : soleil levant`, line 7), *style* (`impressionism`, line 8) and *period* (`first-impressionism`, line 9). The properties of the resources defined for the web page and the painting are specified independently of some *schema.* Nevertheless, RDF gives the possibility to use and to distinguish among different RDF schemas used in RDF descriptions using the *XML namespace* mechanism. Figure 2.2 shows the RDF description illustrated in Figure 2.1 : lines 2 and 3 define two prefixes where the first (`web-page`) contains general properties of HTML pages (e.g. `title, presents, creator`) and the second (`artifact`) specifies properties of cultural artifacts (e.g. `title, style, type, period`). In Figure 2.2, the declarations `<?xml:namespace ns = ... >` is used to define these prefixes. This mechanism is very important since it permits the reuse of existing, distinct RDF schemas within the same RDF

```
1. <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2.   xmlns:web-page ="http://metalab.unc.edu/louvre/namespaces/web-pages"
3.   xmlns:artifact ="http://metalab.unc.edu/louvre/namespaces/artifacts">
4.  <rdf:Description
      about="http://metalab.unc.edu/louvre/paint/monet/first/impression">
5.    <web-page:title>Web Museum: Monet, Claude: Impression: soleil levant
      </web-page:title>
6.    <web-page:presents>
7.      <rdf:Description about="soleil_levant">
8.        <artifact:type>oil painting</artifact:type>
9.        <artifact:title>Impression : soleil levant</artifact:title>
10.       <artifact:style>impressionism</artifact:style>
11.       <artifact:period>first-impressionism</artifact:period>
12.     </rdf:Description>
13.   </web-page:presents>
14.   <web-page:creator>Nicolas Pioch</web-page:creator>
15.  </rdf:Description>
16. </rdf:RDF>
```

Figure 2.2: An RDF description for resource `http://metalab.unc.edu/louvre/paint/monet/-first/impression`.

description, without creating naming conflicts (e.g. `web-page:title`, `artifact:title`).

RDF Schema Specification Language The RDF Schema Specification Language [30] is a declarative language used for the definition of RDF schemas[1] incorporating aspects from knowledge representation models (e.g. semantic nets), database schema definition languages and graph models. It is a simple language of restricted expressive power compared to predicate calculus based specification languages such as CycL [134] and KIF [121].

An RDF schema defines classes and properties which can be instantiated in RDF descriptions. More specifically, an RDF schema is comprised of (1) a *vocabulary*, i.e. a set of class and property names to describe information in a domain, and (2) a set of *semantic relationships* to structure this information.

Classes are organized in hierarchies using the property `subclassOf` which which has the standard semantics of inheritance relationships in object oriented data models. This relationship is denoted in the document using the *prefix* `rdfs` which is associated with the URL `http://www.w3.org/-1999/02/22-rdf-syntax-ns`. For example, the RDF schema illustrated in Figure 2.3 defines class `Man Made Object` (line 4) and its subclass `Iconographic Object` (line 5). It also defines classes `Style` (line 7) and `Period` (line 8).

RDF Schema allows both typed and untyped properties. Properties in our example are typed (i.e. they have a restricted domain and range). In Figure 2.3, property `period` (line 15) is defined between classes `Man Made Object` (line 16) and `Period` (line 17), using the RDF Schema properties `rdfs:domain` and `rdfs:range` respectively.

Summarizing, RDF offers a rich, comparatively simple, graph-based data model and supports the definition of source specific metadata (RDF descriptions) and metadata schemas (RDF schemas).

[1] In the following, *RDF Schema* will denote the specification language used to define RDF schemas.

```
1. <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2.    xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
3.    xmlns:artifact="">
4.   <rdfs:Class rdf:ID="Man Made Object"></rdfs:Class>
5.   <rdfs:Class rdf:ID="Iconographic Object">
6.     <rdfs:subclassOf rdf:resource="#Man Made Object"/></rdfs:Class>
7.   <rdfs:Class rdf:ID="Style"></rdfs:Class>
8.   <rdfs:Class rdf:ID="Period"></rdfs:Class>
9.   <rdf:Property rdf:ID="style">
10.    <rdfs:domain rdf:resource="#Iconographic Object"/>
11.    <rdfs:range rdf:resource="#Style"/> </rdf:Property>
12.  <rdf:Property ID="title">
13.    <rdfs:domain rdf:resource="#Man Made Object"/>
14.    <rdfs:range  rdf:resource="#rdfs:Literal"/></rdf:Property>
15.  <rdf:Property rdf:ID="period">
16.    <rdfs:domain rdf:resource="#Man Made Object"/>
17.    <rdfs:range rdf:resource="#Period"/></rdf:Property>
18. </rdf:RDF>
```

Figure 2.3: An RDF schema for describing cultural resources.

It uses XML for the syntactical representation, exchange, and processing of these metadata.

RDF syntax = XML As already mentioned, RDF uses XML [4] as its syntax, to communicate and process metadata. Metadata communicated between different sources have been defined by different user communities (authorities), and consequently reflect their understanding of the semantics for a given domain or application. Under this perspective, each authority can associate different semantics to the same name used (term or word) in the metadata schema, or use different names to denote the same semantics, giving rise to *naming conflicts* [190] (e.g. *homonyms, synonyms*). RDF uses the XML namespace mechanism [164], which provides a method to distinguish among the different metadata schemas used in source descriptions.

RDF Systems There are a number of systems that deal with the management of RDF data and schemas. One of the most important works in the domain is RDFSuite [180]. The major contributions of this work is the implementation of RDF storage and querying on top of an object-relational DBMS (PostgreSQL). This project is the first to propose an RDF validator and loader (VRP) [13], an RDF description base and a *query language interpreter RQL* [118]. The basic contribution of RQL is that one is able to query both RDF instances (descriptions) and RDF schemas.

Another system is SiLRI [67] which is based on F-logic. Its purpose is the manipulation of RDF descriptions and schemas. In an opposite direction to SiLRI, Metalog [152] uses Datalog to model RDF statements as binary predicates. The last two approaches consider standard logic-based frameworks which are not particularly suited to capture the semi-structure nature of RDF descriptions. The RDF query language proposed by IBM [148] follows a different approach where RDF descriptions are considered as XML documents. The issue is that RDF descriptions are *labeled graphs* with labels on the nodes as well as on the edges which cannot be translated into XML without

loss of semantics.

DAML+OIL DAML+OIL [59] is an expressive language for the definition of metadata vocabularies. It builds on W3C Standards such as RDF and RDFS and extends these languages with much richer modeling primitives. It inherits many aspects from OIL and provides mechanisms which are common to frame-based languages.

Similar to object oriented data models, where the universe of discourse is divided into *objects* and *atomic values*, DAML+OIL introduces the *object* and the *datatype domains*. The object domain consists of instances of classes, and the datatype domain consists of values that belong to XML Schema datatypes.

DAML+OIL Classes DAML+OIL supports the expression of constraints that are much more powerful than those expressed in the RDF Schema Specification Language [30]. First of all, it defines a new subclass of `rdfs:Class`, `daml:Class`. A `daml:Class` class can have more than one superclasses, and can be declared as *disjoint* from another class :

```
<daml:Class rdf:ID="Painting">
  <rdfs:subClassOf rdf:resource="#Man_Made_Object"/>
  <rdfs:subClassOf rdf:resource="#Iconographic_Object"/>
  <daml:disjointWith rdf:resource="#Sculpture"/>
</rdfs:Class>
```

DAML+OIL allows the definition of classes as the *union*, *disjoint union* and *intersection* of other DAML+OIL classes (similar to concept expressions in Description Logic languages). For example, one can define that class `Person` is the disjoint union of classes `Male`, and `Female`.

```
<daml:Class rdf:ID="Person">
  <daml:disjointUnionOf>
    <daml:Class rdf:resource="#Male"/>
    <daml:Class rdf:resource="#Female"/>
  </daml:disjointUnionOf>
</daml:Class>
```

A class can also be defined as an intersection of classes using the `<daml:intersectionOf>` construct.

DAML+OIL Properties DAML distinguishes between *object* and *data type* properties.

```
<daml:ObjectProperty rdf:ID="period">
  <rdfs:domain rdf:resource="#Man_Made_Object"/>
  <rdfs:range  rdf:resource="#Period"/>
</daml:ObjectProperty>

<daml:DatatypeProperty rdf:ID="has_title">
  <rdfs:domain rdf:resource="#Man_Made_Object"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/10/XMLSchema/#string"/>
</daml:DatatypeProperty>
```

The previous example defines two properties. *Object* property *period*, whose domain is class `Man_Made_Object` and range class `Period`. *Datatype* property *has_title* is defined in class `Man_Made_Object` and takes its values in the XML Schema data type `string`. The distinction between object and datatype properties is not possible in RDF Schema. Nor it is possible to use the atomic data types proposed by the XML Schema [26].
Besides the domain and range restriction of properties, DAML+OIL allows the definition of cardinality constraints on properties. For example, one can define that a property is *unique*. For example, if an artifact can have only one title, one can write the following :

```
<daml:UniqueProperty rdf:ID="has_title">
  <rdfs:domain rdf:resource="#Man_Made_Object"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/10/XMLSchema/#string"/>
</daml:UniqueProperty>
```

Another interesting point is that DAML+OIL allows to overwrite a property that has been defined on a superclass, in the definition of its subclasses. For example, imagine that class `Person` is a subclass of `Mammal`. In the latter, the property *has_father* is defined which takes its values in class `Mammal`. If one wants to say that a person can have only a person as her father, then one writes the following :

```
1. <daml:Class rdf:ID="Person">
2.  <rdfs:subClassOf rdf:resource="Mammal"/>
3.  <rdfs:subClassOf>
4.   <daml:Restriction>
5.    <daml:onProperty rdf:resource="#has_father"/>
6.    <daml:toClass rdf:resource="#Person"/>
7.   </daml:Restriction>
8.  </rdfs:subClassOf>
9. </daml:Class>
```

The expression `daml:Restriction` (line 4) defines an anonymous class, namely the class of all 'things' which satisfy the following restriction : the values of the property *has_father* (line 5) of instances of this class, are instances of class `Person` (line 6).

DAML+OIL allows one to define *inverse* properties. For example, one could write the following to state that property *carried_out_by* is the inverse of *carried_out*.

```
<daml:ObjectProperty rdf:ID=''carried_out_by''>
  <daml:inverseOf rdf:resource=''#carried_out''/>
</daml:ObjectProperty>
```

Another interesting feature of DAML+OIL is that one is able to specify that a property is *transitive* using the construct `daml:TransitiveProperty`. For example, one can specify that the property *part_of* is transitive as follows :

```
<daml:TransitiveProperty rdf:ID=''part_of''/>
```

Instances of DAML+OIL classes and properties are defined following the RDF [133] syntax. Here we list some examples of definitions of instances.

```
1. <Person rdf:ID="Theodorus Van Gogh"/>
2. <rdf:Description rdf:ID="Vincent Van Gogh">
3.   <rdf:type>
4.     <rdfs:Class rdf:about="#Person"/>
5.   </rdf:type>
6.   <has_father rdf:resource="#Theodorus Van Gogh"/>
7. </rdf:Description>
```

Resources ''Theodorus Van Gogh'' and ''Vincent Van Gogh'' are instances of class Person (lines 1 and 2-5) respectively. The two resources are related with the property *has_father* (line 6).

There is no unique name assumption for objects in DAML+OIL. Nevertheless, there is a possibility of stating that two resources are the same, using the <daml:sameIndividualAs> or distinct, using the <daml:differentIndividualFrom> construct. The situation is different for datatype values, where the XML Schema Datatype identity is used.

XML Topic Maps Topic Maps, in contrast to RDF and DAML+OIL, have been developed to represent knowledge about resources in general and are not restricted to Web resources. Nevertheless, we present here XML Topic Maps [166] which are used to describe Web information resources.

The role of a topic map is *"to connect information so that we can use it in a more meaningful way than just web hyperlinks. A topic map organizes large sets of information and builds a structured semantic link network over the resources. This network allows easy and selective navigation to the requested information"* [166].
The basic construct in topic maps is *topic*, similar to the notion of *resource* in RDF and DAML+OIL. A topic like a resource can be anything, *"regardless whether it exists or not, whether it is of physical nature or just an idea or expression"* [166]. Similar to RDF, where a resource can be associated with other resources, a topic can be in *association* with other topics. Topics can play different roles in associations and can also contain any number of external references, such as web pages, which supposedly elaborate on a specific topic.

In an XML Topic Map, several topics can be defined. Each topic is associated with an internal identifier, and with *at least* one external identifier. Internal identifiers are names used to refer to this topic. In addition to these identifiers, topics are associated with *base names* which indicate how the topic is presented to the end-user. For example, the topic map shown in Figure 2.4 defines three topics : van_gogh (lines 1-11), starry_night (lines 12-16) and Paris (lines 16-23). The internal identifier of the first topic is van_gogh (line 1) and its base name is Vincent Van Gogh (line 2). It might be the case that the internal identifier and the base name are the same : this is the case for topic Paris (lines 16, 17).

XML Topic Maps allow one to make references to Web pages where information about topics is found. These references are defined by means of *occurrences*. An occurrence can be associated with information which can be external to a topic map, such as references to web sites; or, they can be integrated directly into a topic map as a (short) text. In topic map terminology external resources are managed via the *resourceRef* while internal ones are managed via the *resourceData* construct. For example, topic van_gogh has two occurrences : the first one is associated with the Web page http://www.vangoghgallery.com (line 4). The second is associated with the Web page http://webexhibits.org/vangogh (line 7) and with the internal reference Vincent's Van Gogh Home Page (line 8).

A topic can be an *instance* of one or more topics. For example topic van_gogh is an instance of the topic painter (line 10) and topic Paris is an instance of topics city and capital (lines 21 and 22 respectively). These topics are defined in a similar manner as the ones in Figure 2.4.

```
1. <topic id="van_gogh">
2.   <baseName><baseNameString>Vincent Van Gogh</baseNameString></baseName>
3.   <occurence>
4.     <resourceRef xlink:href="http://www.vangoghgallery.com"/>
5.   </occurence>
6.   <occurence>
7.     <resourceRef xlink:href="http://webexhibits.org/vangogh/"/>
8.     <resourceData>Vincent's Van Gogh Home Page</resourceData>
9.   </occurence>
10.   <instanceOf><topicRef xlink:href="#painter"/></instanceOf>
11. </topic>
12. <topic id="starry_night">
13.   <baseName><baseNameString>Starry Night</baseNameString></baseName>
14.   <instanceOf><topicRef xlink:href="#painting"/></instanceOf>
15. </topic>
16. <topic id="Paris">
17.   <baseName><baseNameString>Paris</baseNameString></baseName>
18.   <occurence>
19.     <resourceRef xlink:href="http://www.paris.org"/>
20.   </occurence>
21.   <instanceOf><topicRef xlink:href="#city"/></instanceOf>
22.   <instanceOf><topicRef xlink:href="#capital"/></instanceOf>
23. </topic>
```

Figure 2.4: Topics

```
1. <association>
2.   <instanceOf>
3.     <topicRef xlink:href="#painted"/>
4.   </instanceOf>
5.   <member>
6.     <roleSpec><topicRef xlink:href="#painter"/></roleSpec>
7.     <topicRef xlink:href="#van_gogh"/>
8.   </member>
9.   <member>
10.    <roleSpec><topicRef xlink:href="#painting"/></roleSpec>
11.    <topicRef xlink:href="#starry_night"/>
12.  </member>
13. </association>
```

Figure 2.5: Associations in Topic Maps

Topics participate in relationships, called *associations*, which are themselves *topics*, in which they play *roles*. Roles are themselves topics. For example, one can define the topic `painted` (to denote for example the relationship between a painter and the paintings she created). In Figure 2.5 an *association* is defined. It is an instance of topic `painted` (lines 2-4) and the members of this association are topics `van_gogh` (line 7) and `starry_night` (line 11). The role of the former is specified by the topic `painter` (line 6) and of the latter by the topic `painting` (line 10).

Comparing RDF or DAML+OIL to XML Topic Maps, one can observe that they have the same principle : everything is a resource for the first two, and everything is a topic for the last. Nevertheless, the distinction between schema and instances (descriptions) is explicit in RDF and DAML+OIL while in XML Topic Maps this is not the case.

2.2.2 HTML Document Annotation Languages and Systems

Document annotation systems follow a different approach for recording the metadata of HTML documents. The basic problem with HTML is that it is a language which does not capture explicitly the semantics of the documents and the only way to retrieve HTML documents is by keyword-based queries or navigation. Web query languages such as WebSQL [155] or WebLog [131] tried to deal with this problem by allowing the combination of querying HTML documents with controlled automated navigation using the outgoing hyperlinks of these pages. For example, WebSQL allows the specification of SQL-like queries on the document contents and structure. One can express a query requesting the documents containing the text "Van Gogh" and all the documents reached from these, by following all outgoing hyperlinks from this page which in their turn contain the text "Oil On Canvas".

Nevertheless, in both cases of indexes and web query languages the problem is that the *semantics* is expressed in the query (i.e. it is explicit in the case of Web languages and implicit in the case of a keyword based query), and not so in the actual data.

Document annotation systems tackled this problem by introducing special *semantic tags* which specify the semantics of the document leading to a more structured Web. For example, consider an HTML page that talks about Van Gogh and its paintings. The part of the document mentioning Van Gogh is annotated by the semantic tag *painter*, and the part containing his paintings can be annotated by the tag `paintings`. These semantic tags originate from *domain* or *application specific*

ontologies which describe the basic notions in the domain of interest, along with basic relations within the domain. These annotations can be considered as content metadata for the resources and can be used for resource discovery.

SHOE [109] is a system supporting a set of Simple HTML Ontological Extensions with its own DTD that specifies *tags* and allowable combinations of those tags, similar to XML and RDF. SHOE supports (i) the *definition of ontologies* (ii) and *annotations* of HTML documents classified by those ontologies.

The principle behind SHOE is that in order to annotate an HTML document, the first thing to be done is to specify the *ontology* which will be used for this purpose. The SHOE *ontology definition* language (ODL) allows the creation of a SHOE ontology which specifies the basic *entity types* or *categories* in the domain of interest. SHOE ODL supports the definition of *n-ary relations* between categories as well as *inheritance* relations between them. It supports also the definition of *inference* rules which are used to deduce *instance* relationships between entities and categories and/or *inheritance* relations between the latter.

SHOE annotations language allows the specification of annotations *within* the HTML document. The ontology used for these annotations is specified in the document by means of a URL, and a *prefix* (similar to XML prefixes), is defined for it.

SHOE annotations are done either for the HTML document or for document fragments. In both cases these annotations are *included* in the document. The annotated part of the document (or the document itself) is considered as a SHOE *instance* and is identified by a *unique key*. This key consists of the document URL, and the position of the text in the document. Each of the instances, is explicitly declared to be an instance of one or more *categories* specified in the ontology. Keys are used to specify *relations* between the SHOE instances.

Annotated HTML pages are gathered by the SHOE crawler. These annotations are extracted from the actual HTML pages and are stored in a Parka knowledge base [83, 196]. Parka is a high performance knowledge representation system whose roots lie in semantic networks and object oriented systems. Data in a Parka knowledge base is stored in a relational database. Once the HTML pages annotated with the SHOE language are gathered they are queried by issuing first order *conjunctive queries*.

Similar to SHOE, *Ontobroker* [86] consists of tools that enhance query access and inference services for Web documents. Ontobroker provides languages to *annotate* Web documents with *ontological* information, to *represent* ontologies and to *formulate* queries. Ontobroker query formalism and document annotation language are oriented towards a *frame-based representation* of ontologies. Ontobroker also allows the specification of *inference rules* (similar to those in Description Logics [27]) that allow to derive facts not present in the actual document. Annotations of HTML pages in Ontobroker are done as in SHOE, i.e. by incorporating special *semantic tags*. Parts of the annotated HTML pages, are considered as objects, instances of defined ontology classes, attribute values and relationships between objects. Annotated HTML pages are gathered, annotations are extracted and stored in the Ontobroker knowledge base.

2.3 Web Data Integration Systems

The problem of *querying* and *integrating* data coming from multiple *heterogeneous* information sources has received significant attention during the last decades. The goal of integration systems is to offer a *uniform* interface to a number of data sources.

As an example, consider the task of providing information about movies from data sources on the Web. There are several sources concerning this kind of information, for example the Internet Movie

Database (providing a list of movies, along with their actors, directors, etc.), Pariscope (providing the playing times of movies for Paris) and several sites providing reviews for these movies. Suppose that a user wants to find the cinemas in Paris playing the movies of Cohen brothers created before 1995, and also the reviews written about each of these movies. None of these sources, considered in isolation, answers this query. Nevertheless, by combining data from these sources one can obtain the answer to this query and even to more complex ones. For the previous query, the user should first look for the movies of Cohen brothers from the Internet Movie Database, then look in Pariscope for the movies playing in Paris, and finally for their reviews on the reviews sites. A user needs to combine information from a number of sources in order to obtain the information she looks for. Moreover, she has to know (i) the structure of the sources and (ii) their query interfaces.

The above problem is the standard problem of *data integration* which becomes more difficult in the Web context due to the increasing number of available, highly heterogeneous information sources. The basic goal of a *data integration* system is to enable its users to concentrate on *what* they want rather than thinking on *how* to obtain their answers. As a result, the user is relieved from the cumbersome task of exploiting each of a possibly large number of sources individually to obtain the information she looks for. More precisely, the challenge of a data integration system is to provide to its users a *unique* view of the underlying data : in this case, users query autonomous and heterogeneous information sources as a *single source* with a *single schema* and a *single query interface*. The task of the system is first to *rewrite* the queries into one or more queries to be evaluated by the sources, then to combine and present the results to the user. To improve the query translation process, as well as the response time, data from the sources can be *materialized* [143] in the user view.

The explosion of the Web as the main road for digital data and information exchange has brought new challenges to data integration. Web sources are *autonomous* and developed in an *independent* way. This autonomicity of the sources results in highly *heterogeneous* sources in *structure*, *semantics* and *access interfaces*. For example, Web data is not just *structured* (relational, object oriented, hierarchical) but also *semi-structured* (e.g. XML data). Furthermore, since the Web is an evolving environment, sources appear and disappear at any time.

The earlier approaches to achieving interoperability between information systems are those of *multi-database* systems [142, 110]. In these systems there is a reasonably small number of sources that are well-structured and do not change over time. Users access the underlying sources by issuing queries in terms of a *single schema* which is the result of the *integration* of the sources schemas. A significant number of techniques for schema integration has been developed in this context [163, 111, 68, 132, 120, 23]. Nevertheless, since Web sources are not necessarily well structured and new sources can appear at any time, standard schema integration methodologies are difficult to apply.

2.3.1 Mediator/Wrapper Architecture

The *mediator/wrapper* architecture proposed by Wiederhold in [216] is a variant of the federated database systems architecture [191] and considers the new challenges set by the Web. In this architecture, the mediator offers a unified view of the data sources in a *specific domain of interest* or a *specific application*. Figure 2.6 illustrates the mediator-wrapper architecture.

Mediator Schema and Source Descriptions The backbone of a mediator is a *global schema* that describes the data that resides in the sources, and exposes the aspects of this data that are of interest to the users. The global schema does not necessarily contain all relations/classes modeled in each of the sources and is *not materialized* : the actual data is found in the sources.

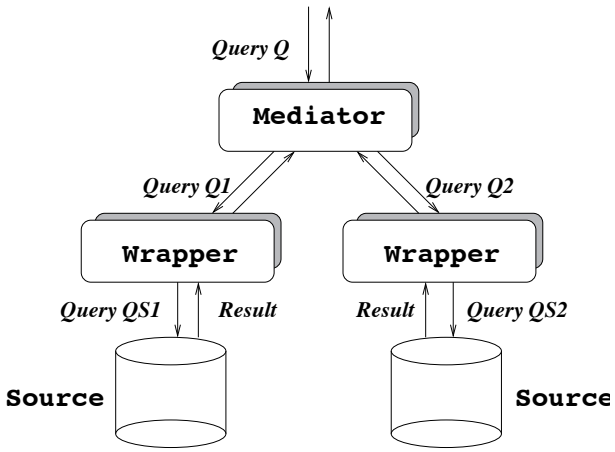

Figure 2.6: Mediator-Wrapper Architecture

Sources are encapsulated by *wrappers* and are described in the mediator by providing their *source descriptions*. These descriptions specify the relationships between the relations of the global schema, and those in the sources schemas. A source description describes the source contents (for example, the source contains paintings with their titles and the techniques applied), constraints (e.g. it contains only paintings of the 19th century), its completeness (e.g. it contains all paintings), and query processing capabilities (e.g. it can answer arbitrary SQL queries, a limited class of XPath queries, or XQuery queries).

There is a number of problems related to the autonomous nature of the sources that must be taken into account when deciding for the language used to define the global schema and describe the sources :

- *overlapping* data between the sources : For example, two sources might both describe Dutch painters of the 18th century. Sources might also contain *contradictory* data. A source for example might specify that Homer was born in one of the Ionian islands, while another might state that he was born in the Aegean island of Chios.
- *semantic mismatches* between the sources : Consider sources which contain XML documents about cultural artifacts. In a source the title of a painting is recorded as an attribute, while in another as an element. Or, in a third source paintings are organized below their respective painters, while in a fourth painters and paintings are recorded as top level elements and horizontal relationships are used to relate them. Furthermore, different names may be used for the same concept. For example, one source may use the term 'artist' and another the term 'creator' to refer to an artist. In a source, the name of a person is a string, while in another is represented as a tuple : (first name, last name).
- *different naming conventions* : A simple example is that various conventions may be used for specifying addresses or dates. Another example, is that one source records the full name of a person, while another contains only her initials. Methods based on textual similarity can be applied to resolve this problem as done in [57].

Query Translation Users issue queries in terms of the global schema. Since the data resides in the sources, the user query must be *rewritten* or *translated* into queries expressed in the local sources schemas.

The requirement is that the rewriting is *sound* (i.e. gives correct answers to the query) and *complete* (i.e. all possible answers extracted from the data sources must be in the result of the rewriting of the query). Moreover, this rewriting must ensure that the sources that do not contribute to the query are not addressed.

Wrappers are proprietary software of sources and their basic purpose is to translate data from the source to a form that is usable by the mediator. For example, if the mediator data model is relational, and the sources are XML, then the wrapper must translate XML data into relational tuples.

Data integration systems are classified according to how the sources are described to the mediator. There exist two approaches : the first, known as *local as view (LAV)*, and the second as *global as view (GAV)*. We first give the framework for data integration in each of these approaches, and then describe in detail some representative systems.

An orthogonal issue to how sources are described in the mediator is the choice of the data model for the global schema (and consequently for source descriptions). The global schema can either be a relational [136, 174], an object oriented [200], an XML DTD [149, 55, 78, 144], an ontology [154], a Description Logic [47, 97] or a semi-structured schema [46].

Most of the integration systems follow the relational paradigm, which has some advantages. In particular the same language can be used as the query language and as the language to define the *source to global schema* mappings (also known as *source descriptions*). Table 2.1 illustrates a classification of the systems based on the following two criteria : (i) the kind of approach (GAV versus LAV) used for the source descriptions and (ii) the data model for the global schema.

	GAV	**LAV**	**Relational/OO**	**Description Logics**	**Semi-Structured**
Information Manifold		■	■	■	
Infomaster		■	■		
PICSEL		■		■	
Xyleme	■	■			■ **(XML)**
Agora		■	■		
Tsimmis	■				■
Nimble	■				■ **(XML)**
MIX	■				■ **(XML)**
MOMIS	■			■	

Table 2.1: Data Integration Systems Classification

2.3.2 Local As View

In the *local as view* [139] approach, the global schema is defined *independently* of the local sources schemas. A source is described in terms of the global schema relations[2], and user queries are formulated in terms of this schema as well. The source relations are defined as (materialized) views over the global schema relations. This approach allows for the modification/publication/deletion of sources without changing the global schema relations.

Let us illustrate an example using a *relational* global schema. Suppose that the domain of interest is *art*. The mediator describes *artifacts* created by *artists*. The global schema relations are

[2]One usually says that in this case, the source is described as a *(materialized) view* of the global schema relations.

ARTIFACT(TITLE, STYLE,NAME,GENRE), ARTIST(NAME,YEAR,SCHOOL) and CRITICS-(TITLE,CRITIC). Relation ARTIST stores all artists. An artist is associated with a name, a year of birth and a school (e.g. Dutch, French). Relation ARTIFACT stores artifacts, each one associated with a title, a style, the artist who created it and a kind (e.g. painting, sculpture). Finally, relation CRITICS stores all critics written for an artifact.

Suppose that two sources are integrated into the mediator : the first is described by relation V_1 and the second by relation V_2. They are both illustrated below. We call V_1 and V_2 *source relations.*

$V_1(name, title)$	$\leftarrow$	ARTIFACT(title,style,name,genre) , ARTIST(name,year,school) , school='Dutch', year $\geq$ 1920, genre='painting'
$V_2(title, critics)$	$\leftarrow$	ARTIFACT(title,style,name,genre) , CRITICS(title,critics) , genre=' painting'

Relation V_1 contains the names of artists (relation ARTIST) and the titles of their artifacts (relation ARTIFACT). The artists contained in the source are Dutch (condition *school = 'Dutch'*) and born after 1920 (condition *year $\geq$ 1920*). Finally the source artifacts are paintings (condition *genre = 'painting'*). The second source, described by relation V_2 contains the title and the critics of paintings (relations ARTIFACT, CRITICS and condition *genre='painting'*). Both views are defined as *rule-based conjunctive queries* over the global schema relations.

The user formulates her queries in terms of the global schema and ignores the source relations (and of course the schemas of the underlying sources). For example, query q illustrated below, looks for the name of Dutch artists born after 1900, the title and the critics of their paintings.

$q(name, title, critics)$	$\leftarrow$	ARTIFACT(title,style,name,genre), ARTIST(name,year,school), CRITICS(title,critics), school='Dutch', year $\geq$ 1900, genre='painting'

In order to obtain answers for the query, this must be *translated* into one or more queries, expressed in terms of the local sources schemas. To find these queries, the initial one must be transformed into a query, called *rewriting* which is formulated in terms of *only* the source relations. In the previous example, neither of the two sources considered in isolation answers query q : the first source returns the name of the Dutch artists and the titles of their paintings, and the second the title of the paintings and their critics. To obtain answers for this query, one must query both sources. In this case one rewriting is :

$$q'(name, title, critics) \leftarrow V_1(name, title), V_2(title, critics)$$

This rewriting is a *rule-based conjunctive* query expressed in terms of only the source relations. Intuitively, it is obtained by replacing each atom in the query by the source relation(s) which contributes to this atom. For example, source relation V_1 contributes with answers to the relations ARTIFACT and ARTIST and V_2 to the relations ARTIFACT and CRITICS. Moreover, V_1 and V_2 satisfy the conditions in the query. V_1 satisfies all the conditions in the query but in a more general case it could satisfy only a subset. If there existed some condition c in V_1 (e.g. *year* $\leq$ 1900) such that the conjunction of c with the conditions in the query is not satisfiable, then V_1 would not have been chosen for the rewriting.

Rewriting Queries Using Views The query translation problem in the LAV approach is known as the problem of *rewriting queries using views* [37, 135, 139]. Informally, the problem is the following. Suppose that we are given a query Q over a database schema, and a set of *view definitions* $V_1, V_2, \ldots V_n$ over the same database schema. Is it possible to answer the query using *only* the answers to the views? This problem has received significant attention because of its relevance to a wide variety of data management problems such as *web-site design* [89], *semantic caching* [64], *data integration* [136, 175, 81] and *data warehouse design* [8, 224, 48, 107]. Authors in [40] distinguish between two problems in *view based query rewriting* : The first concerns *query rewriting* and the second *query answering*.

In query rewriting, given a query and a set of view definitions the goal is to find a *rewriting* that refers only to the views and provides answers to the original query. Given a query Q and a set of view definitions $\mathcal{V} = \{V_1, V_2, \ldots V_m\}$, a rewriting of the query using the views, is a query expression Q' that refers *only* to the views in $\mathcal{V}$. Typically, the rewriting is formulated in the same language used for the query and the views. The example at the beginning of this section illustrates this problem.

In query answering, besides the query and the view definitions, the extensions of the views are also given. The goal is to compute the set of tuples in these extensions i.e. the set of tuples that are in the answer of the query in *all* the databases that are consistent with the views.

Query rewriting has been studied under different assumptions for the form of queries and views. [140] studied the problem of rewriting in the presence of *conjunctive queries* and *views*. The language of rewriting is a union of *conjunctive queries*. Authors proved that the problem of finding a rewriting is NP-hard for two reasons : first because of the possible ways to map a single view into a query and second due to the number of ways to combine the mappings of the different views into the query. The problem of considering binding patterns for queries was studied in [177]. In [195, 56] the problem of rewriting queries with aggregates using materialized views was considered. [24] considers query rewriting in Description Logic. The problem of rewriting queries which use regular path expressions was studied in [34]. The rewriting of recursive queries was considered in [80]. Finally [172] studies the problem of query rewriting for semi-structured views.

The problem of query answering was studied in [6, 38, 35, 36, 100]. The problem of query answering was proven in [6] to be NP-hard and the same result was obtained in [24] for Description Logic queries. A fundamental question in the context of data integration is how to find the rewriting that returns the set of *all* possible answers to the query. This is formulated as the problem of finding *certain answers* [6]. Authors distinguish between the cases where the view extensions are assumed to be *complete* (*closed world assumption*), and the case where the view extensions are assumed to be *partial* (*open world assumption*).

Let us discuss the problem of query rewriting using views. A rewriting can be *equivalent* or *contained* in the initial query. Intuitively, given a set of views and a query Q expressed over the same set of relations, a query Q' is an equivalent rewriting of Q iff for all database instances, the set of answers obtained for Q and Q' are the same. Query Q' is said to be a *contained rewriting* of Q if the set of answers obtained for Q' is a subset of the set of answers obtained for Q. In the LAV approach the rewriting algorithms try to find *maximally contained rewritings*. The difference between equivalent and maximally contained rewritings is that the latter are defined with respect to a *specific* language for the rewritings. Intuitively, there might be a rewriting in a more expressive language that may provide more answers for the query.

The algorithms for testing whether a candidate rewriting is equivalent or contained in the initial query, do not provide a solution on how to *find* the rewriting. These two problems are orthogonal and the last is handled by *query rewriting algorithms*.

2.3.3 Global As View

In the *global as view* approach, the global schema is defined in terms of the local sources schemas[3]. If the global schema and the sources schemas are *relational*, then one can write a *rule-based conjunctive* query over the source relations. This query specifies how to obtain the tuples for the global schema relations. In general, relational global as view descriptions are Horn rules that consider a *single global schema relation* in the head of the rule, and a *conjunction* of atoms over the source relations in the body of the rules. In contrast to the LAV approach, the head of the conjunctive query considers the global schema relations, while the source relations are found in the body of the query.

Consider for example two sources S_1 and S_2. The former contains the names, school, year and place of birth of artists, and the latter contains the title of the objects, their style and the names of the artists who created the objects. A third source, S_3, contains the reviews for artifacts.

ARTIST(name,school)	$\leftarrow$	$S_1(name, year, place, school)$
ARTIFACT(title,name,style)	$\leftarrow$	$S_2(title, style, name)$
CRITICS(title,name,review)	$\leftarrow$	$S_3(title, review)$, $S_2(title, style, name)$

The first description states that tuples for the global schema relation ARTIST(NAME,SCHOOL) are found by source S_1. The second, states that tuples for the global schema relation ARTIFACT(-TITLE,NAME,STYLE) are obtained by source S_2. Finally, tuples for relation CRITICS(TITLE,-NAME,REVIEW) are obtained by a *join* of the tuples originating from sources S_2 and S_3.

User queries are *rule-based conjunctive queries*, formulated in terms of the *global schema relations*. For example, suppose that a query looks for *"the titles, reviews and styles of the objects created by Van Gogh, and his school"*.

$q(title, review, style, school)$	$\leftarrow$	ARTIST('Van Gogh',school), CRITICS(title,review,'Van Gogh'), ARTIFACT(title, 'Van Gogh', style)

Query *reformulation* or *rewriting* in the global as view approach is straightforward : because the relations of the global schema are defined in terms of the source relations, one needs only to substitute the former by their definitions[4]. For the previous query, when substituting the global schema relations by their definitions, the resulting query is :

$q'(title, review, style, school)$	$\leftarrow$	$S_1('Van\ Gogh', year, place, school)$, $S_3(title, review)$, $S_2(title, 'Van\ Gogh', style)$

In q', relation ARTIST is replaced by relation S_1, ARTIFACT by S_2 and finally CRITICS by the conjunction of relations S_2 and S_3. In q' only one appearance of S_2 is kept.

2.3.4 Comparison between the LAV and GAV approaches

As illustrated by the example in Section 2.3.3, the advantage of the global as view approach is the simplicity of query rewriting : it consists of replacing each atom of the query by its definition, the latter expressed in terms of the relations of the local sources.
The major drawback of this approach is its lack of flexibility with respect to the addition/deletion of the sources to the mediator, or the modification of the sources schemas : each update of a source's schema results in an update of the global schema.

[3]One says that the global schema is defined as a *view* over the local sources schemas.
[4]This substitution is called *query unfolding*.

In contrast, in the *local as view*, any modification of the sources (addition/deletion of sources to the mediator, or modification of the source schemas) does not affect neither the global schema relations nor the other source descriptions.

Nevertheless, query rewriting in this approach is complex. As described in Section 2.3.2 this is the problem of *rewriting queries using views*, which is known to be NP-hard [140] in the case of conjunctive queries.

In the context of the Web, it is more appropriate to use the *local as view* approach where the sources are described independently, and any modification of the sources does not affect the global schema.

2.4 Data Integration Systems following the Local as View Approach

A number of data integration projects whether they have been designed for a Web environment or not, have adopted the *local as view* approach for describing sources. Among the most important ones are Information Manifold [136, 174], Infomaster [81], Xyleme [55], Agora [149] and PICSEL [97] which are described in detail.

2.4.1 Information Manifold

Information Manifold [136, 125, 138] tackles the problem of data integration by providing a mechanism to describe declaratively not only the *contents* but also the *query capabilities* of the information sources. The contents of the information source are described by means of *source descriptions* which are used efficiently by the system to prune the information sources that do not provide any answer to a user query, and also to compute executable query plans.

The global schema is relational : it consists of a set of *relations*, augmented with a *class hierarchy*. A source description is a *conjunctive* query over the global schema relations which will be referred as *view* in the following. Information Manifold provides also a declarative mechanism for describing the *query capabilities* of the sources by means of capabilities records. These provide a way to capture the two kinds of source capabilities encountered more often in practice :

- the ability of the sources to apply a number of selections and
- the minimum and maximum number of allowed inputs and the possible outputs of the sources (*binding patterns*).

In general these capabilities define how a source can interact with the other sources, and are used to define *executable query plans*. In the presentation of Information Manifold that is given here, we are restricted to the *global schema data model*, *source descriptions* and *query rewriting algorithm*.

User queries, similar to source descriptions, are *conjunctive* queries expressed in terms of the global schema relations and classes. Query rewriting in Information Manifold is done by the *Bucket* algorithm introduced in [138]. The *MiniCon* algorithm [175] is an improved version of the Bucket algorithm.

As with other data integration systems, Information Manifold is not concerned with issues such as updates and transaction but only with *querying of the information sources*. In the following, we present the data model, source descriptions and the Bucket and MiniCon algorithms.

Data Model of the Global Schema Information Manifold uses the *relational model* for the global schema, augmented with certain features from *description logic languages*. Those features

are useful for *describing* and *reasoning* about the contents of information sources. The data model is based on *relations* of n-arity, and *classes* that form a *class* hierarchy which defines a partial order. Each class is associated with a set of *attributes*. A class can inherit attributes from its superclasses and attributes can be *single-valued* or *multi-valued*.

The extension of a relation is a set of tuples, while the extension of a class is a set of objects, where each object has a unique object identifier. The values of the attributes of a relation or a class, can be either atomic values or object identifiers. Multiple instantiation for classes is allowed : an object can belong to different classes even if those are not related by the class hierarchy. Finally, it is possible to declare that two classes are *disjoint*.

Classes and *class attributes* are encoded as *unary* and *binary* relations respectively[5] :

- If C is a class, and x is an identifier for some object, instance of C, then $\langle\ x\ \rangle$ is a tuple in the unary relation C (i.e. $\langle\ x\ \rangle\ \in C$).
- If A is an attribute defined on class C, and x is the identifier of an object instance of C, then $\langle x, y\rangle\ \in A$ if $x.A = y$ (y is the value for x of attribute A).
- If C is a subclass of C' then $C \subseteq C'$.
- To capture the partial order between classes, the *inclusion dependency* between their corresponding relations is expressed. If two classes C, D are defined to be disjoint, then $C \cap D = \phi$.

For our cultural example, the global schema classes are illustrated in Table 2.2.

Class	**Subclass Of**	**Attributes**	**Disjoint from**
Actor		*carried_out*	
Person	Actor	*has_name*, *carried_out*	
Event		*took_place_at*, *took_place_in*	Actor
Activity	Event	*has_produced*, *took_place_at*, *took_place_in*	
Man_Made_Object		*has_title*, *created_by*	Actor
Place		*placeName*, *city*	Actor
Date		*year*, *month*, *day*	Actor

Table 2.2: Examples of global schema relations in Information Manifold

To decide whether two objects retrieved from different sources correspond to the same real world entity, each object (instance of some class), is associated with a *unique identifier*. This unique identifier is calculated out of the value of some attribute (for example, the name of a person). [136] does not provide any additional information on neither how these object identifiers are created nor how the attributes, out of whose values the identifiers are created, are determined.

User Queries Users formulate their queries in terms of the global schema relations. A user query is a *conjunctive* query over the mediated schema relations, using built-in comparison predicates

[5]For each class C, a relation is defined with the same name as the class. For each attribute of the class, a *binary* relation is created.

($\leq$, $\geq$, $<$, $>$). In general, a query is of the form

$$Q(\bar{X}) \leftarrow R_1(\bar{Z}_1), \ldots R_n(\bar{Z}_n), C_Q$$

where the R_i's are the global schema relations, C_Q is a conjunction of comparison predicates of the form $u\theta v$ where $u, v \in \cup_{1 \leq i \leq n} \bar{Z}_i$ or v is a constant, and $\bar{X} \subseteq \cup_{1 \leq i \leq n} \bar{Z}_i$.

For example the query illustrated below, looks for *"the names of persons, the titles and year of creation of the objects they created after 1800"*.

$q(N,T,Y)$	$\leftarrow$	$Person(P), has\_name(P,N), carried\_out(P,A),$ $Activity(A), has\_produced(A,M),$ $Man\_Made\_Object(M), has\_title(M,T),$ $created\_by(M,E), Event(E),$ $took\_place\_in(E,D), Date(D), year(D,Y),$ $Y \geq 1800$

Table 2.3: User Query

Source Descriptions A source description in Information Manifold is expressed as a *conjunctive query* over the global schema relations. The head of the conjunctive query is called a *source relation* (or *view*).

Let us consider again the cultural example. The definitions of the source relations V_1, V_2 and V_3 are illustrated in Table 2.4.

$V_1(n,t)$	$\subseteq$	$Person(p), has\_name(p,n),$ $carried\_out(p,a), Activity(a),$ $has\_produced(a,m), Man\_Made\_Object(m),$ $has\_title(m,t)$
$V_2(t,y)$	$\subseteq$	$Man\_Made\_Object(m), has\_title(m,t)$ $Event(e), created\_by(m,e), took\_place\_in(e,d), Date(d),$ $year(d,y), y \geq 1850$
$V_3(d)$	$\subseteq$	$Event(e), took\_place\_in(e,d), Date(d), year(d,y), y \leq 1700$

Table 2.4: Source Descriptions

The first source relation $V_1(n,t)$, returns the names of persons, and the title of the objects created by them. $V_2(t,y)$ returns the title, and the year of creation of objects which have been created after 1850. Last, $V_3(d)$ returns the dates of events occurred before 1700. The connective $\subseteq$ is used in the source descriptions to underline the fact that the source provides only a subset of the tuples that satisfy the global schema relations instead of the connective $\leftarrow$ used in the user queries.

The Bucket Algorithm Given a user query expressed in terms of the global schema relations, and a set of views, the Bucket algorithm tries to rewrite the user query into a query that is expressed only in terms of the source relations.

The algorithm finds rewritings by pruning the sources that do not contribute with answers to the user query. Cases where a source cannot provide answers to a query are the following :

- the conjunction of the query constraints and those expressed in the source description is not satisfiable. For example, the query requests the objects created after 1900 and the third source contains objects created before 1700;
- the query requests instances of some relation R and the source returns instances of a relation R' where R and R' are disjoint. For example, the query requests instances of relation *Event* and the source returns instances of relation *Person*.
- last, the query requests values of some attribute, but the source does not return these values. For example, consider a source which contains the names of persons but does not export this information in its description. Hence, this source cannot be used for queries that request the names of persons.

The Bucket algorithm is a two phases algorithm :

1. In the first phase it considers each subgoal of the query in isolation and tries to find which are the source relations that provide answers to it. These relations are added in the *bucket* of the query subgoal.
2. In the second phase, it finds a set of *conjunctive query rewritings*. Each such rewriting is a conjunctive query that includes *one conjunct* from each bucket. Each of these rewritings represent one way to obtain answers for the query.

   The result of the algorithm is the *union* of the conjunctive query rewritings. For each of this conjunctive rewritings it must be verified that :

   (a) it is *contained* in Q and

   (b) that after adding the comparison predicates in Q, it is still contained in Q.

Example Let us illustrate the bucket algorithm by an example. Consider the query illustrated in Table 2.3, and the source relations in Table 2.4.

Calculating the buckets The buckets and their contents are illustrated in Table 2.5. For each bucket, we give (i) the views and (ii) the mappings unifying the variables in the view subgoal and the variables in the query subgoal.

In the first step, the algorithm considers the bucket for the query subgoal $Person(P)$. V_1 contributes to this subgoal since (i) it contains the relation (subgoal) $Person(p)$ and (ii) there is a mapping unifying the variables in the query and view subgoals. This mapping is $\{P \rightarrow p\}$. So, V_1 is added in the bucket for the query subgoal $Person(P)$.
In a similar manner, V_1 is added in the buckets for the query subgoals $carried\_out(P, A)$, $Activity(A)$, and $has\_produced(A, M)$.

Consider now the query subgoal $has\_name(P, N)$. For this subgoal there is a mapping between its variables and the variables of the subgoal $has\_name(p, n)$ in V_1 : $\{P \rightarrow p, N \rightarrow n\}$. Moreover,

$Person(P)$:	$[V_1, \{P \rightarrow p\}]$
$has\_title(M,T)$:	$[V_1, \{M \rightarrow m\}]$
	$[V_2, \{M \rightarrow m\}]$
$has\_name(P,N)$:	$[V_1, \{P \rightarrow p, N \rightarrow n\}]$
$Event(E)$:	$[V_2, \{E \rightarrow e\}]$
	$[V_3, \{E \rightarrow e\}]$
$Activity(A)$:	$[V_1, \{A \rightarrow a\}]$
$created\_by(M,E)$:	$[V_2, \{M \rightarrow m, E \rightarrow e\}]$
$carried\_out(P,A)$:	$[V_1, \{P \rightarrow p, A \rightarrow a\}]$
$Man\_Made\_Object(M)$:	$[V_1, \{M \rightarrow m\}]$
	$[V_2, \{M \rightarrow m\}]$
$took\_place\_in(E,D)$:	$[V_2, \{E \rightarrow e, D \rightarrow d\}]$
	$[V_3, \{E \rightarrow e, D \rightarrow d\}]$
$Date(D)$:	$[V_2, \{D \rightarrow d\}]$
	$[V_3(d), \{D \rightarrow d\}]$
$has\_produced(A,M)$:	$[V_1, \{A \rightarrow a, M \rightarrow m\}]$
$year(D,Y)$:	$[V_2, \{D \rightarrow d, Y \rightarrow y\}$

Table 2.5: Buckets for the subgoals of query q

variable N is returned by the query, and V_1 returns values for N (notice that the head of its definition contains variable n to which N is mapped to). For the query subgoal $Man\_Made\_Object(M)$, V_1 and V_2 are added in its bucket through the mapping $\{M \rightarrow m\}$.

Consider now the subgoal $year(D,Y)$. V_2 contributes to this subgoal by the mapping $\{D \rightarrow d, Y \rightarrow y\}$. After the unification of the variables, the algorithm tests the predicates $Y \geq 1800$ and $y \geq 1950$. These are consistent, therefore V_2 is considered for this query subgoal. A subtle point here is that V_2 is considered, because variable Y (requested by the query) appears in the head of the V_2. This is not the case for V_3. Although there is a mapping between the query variables in the subgoal and the corresponding subgoal in the view, variable Y (requested by the query) does not appear in the head of V_3 (in other words, the view does not contribute to this variable). So, V_3 is not considered for the bucket of this query subgoal.

Computing conjunctive query rewritings In the second phase the *conjunctive query rewritings* are computed where a rewriting includes one conjunct (i.e. view) from each bucket. For example, for the buckets shown in Table 2.5, when the first element of each bucket is considered[6], then the rewriting q' is obtained :

$$q'(N,T,Y) : -V_1(N,T), V_2(T,Y)$$

Another rewriting is $q''(N,T,Y) : -V_1(N,T), V_2(T,Y), V_3(D)$. But if we consider the conditions of the views V_2 and V_3, we see that their conjunction is *not satisfiable*. When these views were considered in isolation by the Bucket algorithm their conditions were not found to be unsatisfiable. Consequently, q'' is not considered as a rewriting.

The bucket algorithm produces *maximally-contained rewritings* as is the case in data integration scenarios using the LAV approach. The strength of the Bucket algorithm is that during the creation of the buckets, it prunes significantly the number of sources that need to be considered for the

[6]After keeping one appearance for each view in the rewriting.

candidate query rewritings. Checking whether a view can be added to a subgoal can be done in time polynomial in the size of the query and view definition when the predicates involved are arithmetic comparisons. Nevertheless, the algorithm has certain drawbacks :

- the number of rewritings to be considered in the second phase (i.e. the Cartesian product of the views in the buckets) can be rather large. Moreover, for each of these rewritings the algorithm performs a *query containment* test. To do that, it replaces the views in the rewriting by their definitions, and then tests whether the query obtained is contained in the original query. The testing problem is Π^2-complete [127], though only in the size of the query and the view definition.

- since it examines each query subgoal in isolation, it does not examine how the views interact. In this sense, in the second step, it produces candidate rewritings which will not be considered.

MiniCon Algorithm The MiniCon algorithm [175, 174] addresses the limitations of the Bucket algorithm.

The Bucket algorithm prunes information sources which are either irrelevant to the query, or the do not satisfy the query's conditions. Nevertheless, in the second step of the algorithm, a large number of view combinations must still be calculated. Each of these combinations forms a conjunctive rewriting for which it must be proved that it is contained in the initial query. Another point is that at the last step of combining the views it might be that the conditions expressed in two distinct views are mutually inconsistent. The MiniCon algorithm addresses the above limitations by changing its perspective : instead of building rewritings by combining views for each of the query subgoals, it considers how *each* of the variables in the query interacts with the available views. Consequently, in the second phase, the MiniCon algorithm needs to consider drastically fewer combinations of rules.

Let us consider again the cultural example. Consider a user query which requests *"the year of creation of objects whose title is 'Madonna' "*. This query is illustrated in Table 2.6.

$$q(Y) \leftarrow Man\_Made\_Object(M), has\_title(M,'Madonna'), Event(E), created\_by(M,E), took\_place\_in(E,D), Date(D), year(D,Y)$$

Table 2.6: User Query

Consider now the views of the sources, which are illustrated in Table 2.7.

$V_1(t)$	$\subseteq$	$Man\_Made\_Object(m)$, $has\_title(m,t)$
$V_2(y)$	$\subseteq$	$Event(e)$, $took\_place\_in(e,d)$, $Date(d)$, $year(d,y)$

Table 2.7: Source Descriptions

The bucket algorithm would consider view V_1 in the bucket of subgoals $Man\_Made\_Object(m)$ and $has\_title(m,t)$. View V_2 will be considered in the bucket of the subgoals $Event(e)$, $took\_place\_in(e,d)$, $Date(d)$ and $year(d,y)$. The problem is that none of the views is useful for a rewriting of this query. The reason is that the views do not contain some variable on which one could join the results of the views (the case of variables M and E).

The MiniCon algorithm starts like the bucket algorithm, considering which are the views that contain subgoals that correspond to subgoals in the query. However, when the algorithm finds

a partial mapping between the variables in the subgoal g in the query and a subgoal g' in the view, then it changes perspective and it examines the variables in the query. The algorithm then considers the join predicates in the query, and finds the minimal additional set of subgoals that must be mapped to subgoals in the set of views, given that g is mapped to g'. The join predicates are determined by the multiple occurrences of the same variable in the query. The set of subgoals given a mapping plus the mapping is called an *MCD* (MiniCon description). An MCD can be seen as a generalized bucket. In the second phase, the MCDs are combined to produce the query rewritings. For the previous example, the algorithm does not create an MCD for V_1 since it cannot apply the join predicates of the query.

The key advantage of the MiniCon algorithm is that it considers in the end fewer combinations than the Cartesian product of the views in the buckets created by the Bucket algorithm. The algorithm outperforms the bucket algorithm and the inverse rules algorithm [81]. Moreover, experiments in [174] demonstrate that the algorithm scales up to hundreds of thousands of views.

2.4.2 Xyleme : A Web Scale XML Repository

Xyleme is a dynamic warehouse for the XML data of the Web [222]. The research directions that have been studied in the Xyleme project are : (i) *efficient storage* of huge volumes of XML data [117], (ii) *query processing* [10, 55, 208] with sophisticated indexes at the XML element level; (iii) *data acquisition* [160] strategies to build the XML repository, (iv) *change control* [153] of XML documents and (v) *semantic data integration* [182] to free the user from knowing the individual XML resources for expressing her queries. We are interested in the *query processing* aspects of the Xyleme system and more specifically in the *source description language* and the *rewriting algorithm.*

Xyleme Views One of the objectives of Xyleme is to provide a *single access* to all the XML documents stored in the Xyleme repository, by hiding their heterogeneities to the end-user[7]. As in standard data integration systems, the objective here is to relieve the user from querying each heterogeneous structure to obtain the answers to her queries.

The novel point of Xyleme's architecture is the following : XML documents stored in the Xyleme repository do not only have heterogeneous structures but are also associated with different domains of interest. For example, in the Xyleme repository documents from the domains of medicine, culture, mathematics could be stored and queried. XML documents that belong to a specific domain of interest are organized in a *cluster*. In Figure 2.7, two clusters are illustrated. The first collects documents in the art domain and the second collects documents in the domain of cinema.

XML documents within a *cluster* are instances of different XML DTDs, called *concrete DTDs*. There might exist a large number of such DTDs in a single cluster. Concrete DTDs in Xyleme are *tree structures* : no distinction is made between *elements* or *attributes* and no constraints (i.e. like occurrence indicators) are considered.
In the rightmost part of Figure 2.8 two concrete DTDs are shown from the "Art" cluster :

- the first (DTD_1) describes painters and their paintings. Each painter has a biography and a name. Each painting has a name, a year of creation, a technique and a location where it is exposed. Paintings are organized below their respective painter.

[7]The objective of the initial Xyleme project was to crawl the Web for all the XML documents and store them in the Xyleme repository. Because of the eventual size of this repository, data integration was a crucial issue. Currently, the Xyleme repository is used for storing XML documents associated with a *single* application and hence data integration is no longer a problem.

- the second (DTD_2) describes artifacts (works of art), where each artifact is associated with an artist, a title, a year and a gallery. An artist has a name, and is associated with the period during which she created the artifacts.

For an XML document conforming to these DTDs, each element can have zero or more children nodes of the same type, e.g. a painting can be associated with zero or more techniques.

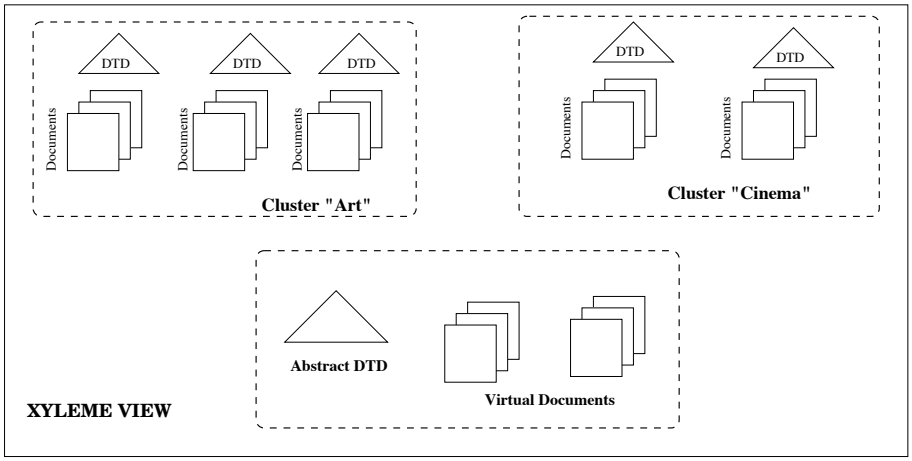

Figure 2.7: Xyleme Clusters

For clusters which are semantically related, a *view à la Xyleme* is defined :

- the *domain* of this view is the set of documents that belong to the clusters;
- the *view schema*, similar to the *global schema* of a mediator, is an *abstract DTD*. It is a *tree structure* of *concepts* rather than elements or attributes. An abstract DTD is defined for a number of clusters, and hence for a number of concrete DTDs (recall that a cluster contains numerous concrete DTDs). It is used to represent an abstract structure in a number of domains of interest for example **Culture**, **Tourism**, **Medicine** and hides the structural and semantic heterogeneities of the documents in the underlying clusters;
- finally, a view definition is a set of *mappings* between *abstract paths* and *concrete paths*. Paths in the abstract and concrete DTDs provide the *context* for the nodes in the tree.

In Figure 2.7, for the two clusters one view is defined, where its schema is the abstract DTD illustrated in Figure 2.8. Abstract DTDs are not materialized, only the concrete ones are.

In the abstract DTD illustrated in Figure 2.8, all **Person** concepts correspond to the same notion of person. But, each appearance of a concept is interpreted in the context of its parent node in the tree. For example, the **Person** concept below the **Man_Made_Object** concept represents the creator of the painting, while the **Person** concept below the **Movie** concept represents the director of the movie. In a similar manner, the **Place** concept found below the **Man_Made_Object** concept corresponds to the location where the object is exposed, while the same concept appearing below the **Event** concept corresponds to the place where the event took place.

View Definition Language In contrast to standard data integration approaches where a view is generally a query, in Xyleme a view is defined in terms *mappings* between *abstract paths* and

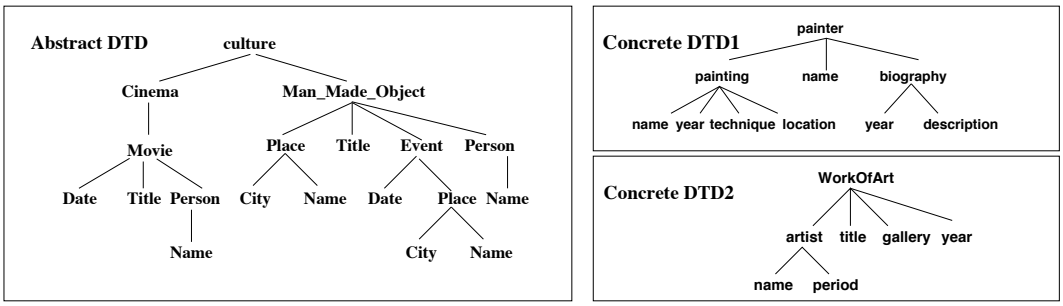

Figure 2.8: An abstract and concrete DTDs for the Cultural Domain

concrete paths. A concrete DTD is described by a set of *abstract to concrete path mappings* where abstract/concrete paths are specified in the abstract and concrete DTDs respectively.

Example Consider the abstract DTD and the concrete DTD_1 illustrated in Figure 2.8. The latter is mapped to the former by means of the mappings illustrated in Table 2.8. The concrete DTD_2 illustrated in Figure 2.8 is mapped to the abstract DTD by means of the mappings illustrated in Table 2.9. From this point and for the examples, abstract paths are represented in **this** font and concrete paths in this font.

P_1	:	**culture/Man_Made_Object**	$\rightarrow$	painter/painting
P_2	:	**culture/Man_Made_Object/Title**	$\rightarrow$	painter/painting/name
P_3	:	**culture/Man_Made_Object/Person**	$\rightarrow$	painter
P_4	:	**culture/Man_Made_Object/Place**	$\rightarrow$	painter/painting/location
P_5	:	**culture/Man_Made_Object/Person/Name**	$\rightarrow$	painter/name

Table 2.8: Abstract to Concrete Path Mappings for Concrete DTD_1

S_1	:	**culture/Man_Made_Object**	$\rightarrow$	WorkOfArt
S_2	:	**culture/Man_Made_Object/Title**	$\rightarrow$	WorkOfArt/title
S_3	:	**culture/Man_Made_Object/Person**	$\rightarrow$	WorkOfArt/artist
S_4	:	**culture/Man_Made_Object/Place**	$\rightarrow$	WorkOfArt/gallery
S_5	:	**culture/Man_Made_Object/Person/Name**	$\rightarrow$	WorkOfArt/artist/name

Table 2.9: Abstract to Concrete Path Mappings for Concrete DTD_2

Mapping P_3 in Figure 2.8 states that the abstract concept **culture/Man_Made_Object/-Person** (i.e. the creator of an object) corresponds to the concrete concept painter in DTD_1.

Abstract and concrete paths in a mapping satisfy the following *conditions* :

- abstract/concrete paths are always specified from the *root* of the abstract/concrete DTD respectively. For example, we cannot specify the abstract path **Person/Name** neither the concrete path artist for DTD_2;
- abstract/concrete paths use *always* the *child* axis;
- finally, if an abstract path is mapped to a concrete path, then there does not exist a prefix of the abstract path, mapped to two distinct prefixes of the concrete path. In short, there is

a single mapping between nodes in an abstract path to nodes along some concrete path. For example, consider the two concrete paths painter and painter/influenced_by/author. Then if the abstract path **culture/Man_Made_Object/Person** is mapped to the concrete path painter, it is not possible to map the same path to the concrete path painter/influenced_by/-author. This restriction is done basically for reducing the complexity of the rewriting algorithm presented below.

Queries A user can formulate queries in terms of abstract DTDs or in terms of concrete DTDs published in the Xyleme system. Queries specified in terms of the abstract DTDs are simple *tree queries*. On the other hand, queries can be formulated in terms of concrete DTDs using the X-OQL [9] query language, a full fledged XML query language, consistent with the requirements published by the W3C XML Query Working group [211] and similar to several languages for semi-structured data.

In this presentation of the Xyleme system we are concerned with the query language used to formulate queries over the abstract DTDs. Consider for example the query illustrated in Figure 2.9(a) which requests *"the title of the objects exposed in the Orsay museum and their artists"*. Variable binding paths are a restricted case of XPath location paths [52] where only the *child* axis is used[8]. More specifically, the binding paths of the query variables are paths which navigate *downwards* the abstract DTD. Conditions can be specified for these paths by using the full text predicate *contains*. No joins are allowed, i.e. if variable x is bound to path a/b and y to c/d one cannot express the fact that $x = y$. Furthermore, no restructuring, aggregate operations are allowed and the language has no subqueries.

The structure specified in the query is used as a means to *filter* documents. For the previous query the collection named *culture* contains documents conforming to a large number of heterogeneous XML DTDs. The previous query concerns only those which have the structural pattern specified in Figure 2.9(b).

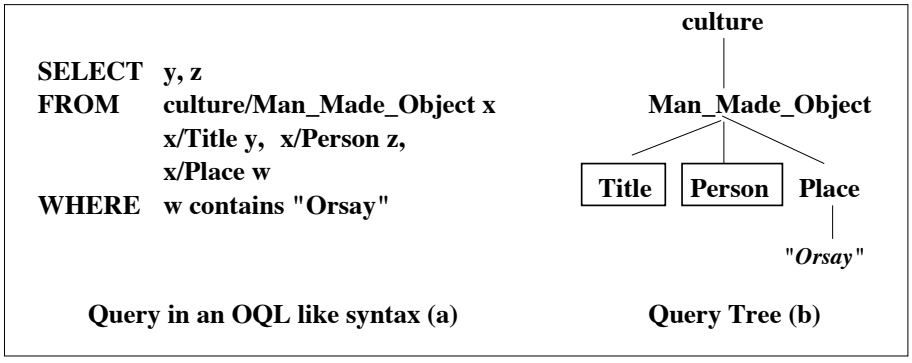

Figure 2.9: A user query

Query rewriting Abstract DTDs are not materialized. Only the concrete ones are. In order to provide answers to a user query formulated in terms of an abstract DTD, the query needs to be *rewritten* to concrete queries expressed in terms of the concrete DTDs that belong to the domain of the former. This is the standard problem of *rewriting queries using views* presented earlier, where

[8]No predicates are allowed and no functions are used.

a view in Xyleme is a set of *abstract-to-concrete path mappings*. The answer to a user query is the union of answers obtained by the concrete queries.

Query rewriting in Xyleme consists essentially of *translating* the abstract paths occurring in the user query to the concrete paths of some concrete DTD as follows :

- In the first step the query is decomposed into paths which start from the root and arrive to a leaf node;
- in the second step, for each abstract query path a, a set of concrete paths $\{c_1, c_2, \ldots c_n\}$ is selected such that for all c_i, there exist an abstract-to-concrete-mapping $a \rightarrow c_i$. The concrete paths $c_i, 1 \leq i \leq n$ belong to different concrete DTDs. The association of an abstract path a_i to a concrete path c_i is called a translation. We have to note here that the algorithm examines at the same time all concrete DTDs, and not one by one. This is an optimization of the algorithm and is possible due to the presence of adapted index structures for the efficient exploitation of the mappings. At this step, among the possible ways of combining the concrete paths the algorithm selects those that :
  1. form a subtree of a concrete DTD and
  2. preserve the *prefix* relationship between the paths in the abstract DTD. Take for example a translation which considers the abstract-to-concrete mapping $a_i \rightarrow c_i$. If a_j is a prefix of a_i and $a_j \rightarrow c_j$ is an abstract-to-concrete mapping, this is considered as a translation of a_j only if c_j is a prefix of c_i.

  The result of this last step is the *union* of the concrete queries produced by the valid combinations of the concrete paths (which belong to the same concrete DTD).

The rewriting algorithm proceeds as follows : after all abstract paths of the query have been identified, the leftmost path is considered (an order is assumed for the query paths). The algorithm finds a concrete path to which this abstract path is mapped to from the set of the abstract-to-concrete-path mappings. If such a concrete path exists, then it is considered as a *translation* of the abstract path. Then, the algorithm examines in turn all the abstract query paths which are *prefixes* of the examined abstract path. It examines each prefix in turn and tries to find a concrete path to which this is mapped to in the set of the mappings. If a concrete path is found, then the algorithm verifies if this is a *valid translation*.

A translation is valid, if the prefix relationship between two abstract paths also holds between their translations (i.e. the concrete paths). In other words, the algorithm does not accept translations where the prefix relationship between the abstract paths does not hold for their translated concrete paths.

After all prefixes are considered, then the algorithm examines, as previously, the sibling abstract leaf path. As far as the prefixes of this path are concerned, the algorithm proceeds as follows : it examines all of them in turn, but stops when it finds an abstract path which has already been considered. The algorithm ends, when all abstract paths are considered or when a translation which is not valid is found.

Example Let us illustrate the algorithm for the query illustrated in Figure 2.9. We present the rewriting for the concrete DTD_1 first and then for DTD_2. The result of the first step of the algorithm is the set of abstract paths illustrated in Table 2.10. The order of the appearance of the paths in the table is from the leftmost to the rightmost path in the query tree.

A_1	**culture/Man_Made_Object/Title**
A_2	**culture/Man_Made_Object**
A_3	**culture/Man_Made_Object/Person**
A_4	**culture/Man_Made_Object/Place/**

Table 2.10: Abstract paths for the user query in Figure 2.9

Let us now run the rewriting algorithm for the concrete DTD_1. The algorithm considers first the leftmost path, and finds a concrete path to which this abstract path is mapped to. The abstract path A_1 is translated into the concrete path P_2 from the mappings of Table 2.8.
Then, the abstract path A_2 is considered, which is a prefix of the abstract path A_1. It is translated to the concrete path P_1 of Table 2.8. At this point, the algorithm checks whether this translation is a valid one. More specifically, it checks that the prefix relationships between abstract paths hold between their translations. This is a *valid translation* since the P_1 is a prefix of P_2 (i.e. path painter/painting is a prefix of painter/painting/name).
In the third step the abstract path A_3 is considered. From the set of mappings in Table 2.8 this is mapped to the concrete path P_3. This is not a valid translation : the concrete path P_1 (painter/painting) *is not* a prefix of the concrete path P_3 (painter). The algorithm stops here and the remaining abstract paths are not considered.

Let us now illustrate the rewriting of the same query for the concrete DTD_2. The mappings are illustrated in Table 2.9. The algorithm considers first the leftmost path, and finds a concrete path to which this abstract path is mapped to : the abstract path A_1 is translated into the concrete path S_2 from the mappings of Table 2.9.
Then, the abstract path A_2 is considered, which is a prefix of A_1. It is translated to the concrete path S_1 of Table 2.9. This translation is a valid one, since the prefix relationship between the abstract paths A_2 and A_1 holds also for their translated concrete paths.
In the third step the abstract path A_3 is considered. From the set of mappings in Table 2.9 this is mapped to the concrete path S_3. Again, this translation is a valid one since the prefix relationship between the abstract paths A_2 and A_3 holds also for their translated concrete paths. Last, abstract path A_4 is considered which is translated into the concrete path S_4.

Discussion : The complexity of the algorithm is *linear* in the number of mappings for an abstract path. This is mainly due to the restriction that *only one* translation is considered for an abstract path : if an abstract path is mapped to more than one concrete paths, only one of the latter ones is considered for a translation. If the abstract path does not lead to a leaf node, the "longest" concrete path is considered. This might lead to cases where no further translations can be found. Moreover, the algorithm looses a number of answers, due to the constraint of preserving the prefix relationship between the abstract and the concrete paths. Due to this restriction the Xyleme rewriting algorithm *is not complete.* In the case where few answers are obtained there is a *relaxation* phase. There are two choices :

1. first, the constraint on the preservation of the prefix relationship between the abstract and their corresponding concrete paths is relaxed. In this case, the complexity of the rewriting algorithm is augmenting : all combination of mappings that have the same concrete root must be considered.

2. second, all paths leading to conditional nodes are removed, and only the paths that lead to projection nodes are considered.

Conclusions Xyleme is a data warehouse aiming to store the XML data of the Web. The size of data is an important factor which determined the choices of (i) the *global schema* (abstract DTD) (ii) the *view definition language*, and (iii) the *query rewriting algorithm*. Let us examine each one of the previous choices in turn :

Xyleme Global Schema : The Xyleme global schema is an *abstract DTD* i.e. a tree of concepts rather than an XML tree (tree of elements and attributes). Each concept is interpreted in the context of its parent concept. Hence, to capture the different contexts in which a single concept appears, this must appear several times in the abstract DTD. Consider for example the abstract DTD illustrated in Figure 2.8. Notice that the concept **Person** appears two times : below the **Movie** concept to denote the director of the movie and below the **Man_Made_Object** concept to denote the creator of the object.
Notice that the two **Person** concepts have exactly the same structure (i.e. represent actors) but they are repeated in the XML DTD to capture the different roles that an actor may play with respect to the context in which she appears.
In the Web context, where an abstract DTD summarizes a number of different but closely related structures, the repetition of the same (or similar structures) might lead to a considerable overload of the DTD.

This problem is related to the absence of *generalization/specialization hierarchies* which forces the repetition of structures common to several concepts. For example, if the abstract DTD contains the concepts **Painting** and **Sculpture** which have the same structure, the latter must be repeated in the XML DTD. There are also the following two issues : (i) precision of mappings and (ii) formulation of queries.

Consider the abstract DTD illustrated in Figure 2.8. It contains concept **Man_Made_Object** which collects all cultural artifacts (paintings, sculptures etc.). In this case, one must map different kinds of artifacts found in a source to a single concept. A solution would be to add appropriate concepts for the specific kinds of artifacts in the abstract DTD. For example one might add the abstract paths **Man_Made_Object/Painting** and **Man_Made_Object/Sculpture** for the concepts **Painting** and **Sculpture.**
A source can then map paintings to the first path and sculptures to the second path. The tree of concepts found below **Man_Made_Object** must be repeated for the new concepts added. Hence, *generalization/specialization* hierarchies can be 'simulated' in the abstract DTD. Nevertheless, the semantics of such hierarchies (commonality of structures and set containment) are not explicit in such representations. Consider now the formulation of queries. For the previous case, if a user looks for objects in general, then three queries must be specified : one using the abstract path **Man_Made_Object/Painting**, a second using the abstract path **Man_Made_Object/Sculpture** and finally a third one which uses the abstract path **Man_Made_Object**. If the user specifies only the last query, then the rewriting algorithm will not consider the concrete DTDs that have used the other abstract paths in their mappings.

View Definition and Rewriting Algorithm : Abstract and concrete paths use only the *child* axis and are defined from the root of the abstract/concrete DTD. This assumption does not restrict the sources that can be described[9]. Consider again the abstract DTD and the concrete DTD_1, both illustrated in Figure 2.8. Notice that in the abstract DTD, creators of artifacts are found below the artifacts that have created. In the concrete DTD, paintings are organized

[9]Except if elements in the concrete DTD are organized as top level elements and horizontal relationships are used to connect these elements.

below their painters. Mappings P_1 and P_3 illustrated below, associate abstract with concrete paths.

$$\begin{array}{lll} P_1 : \textbf{culture/Man\_Made\_Object} & \rightarrow & \text{painter/painting} \\ P_3 : \textbf{culture/Man\_Made\_Object/Person} & \rightarrow & \text{painter} \end{array}$$

Consider the following query which looks for objects.

Q:	**select**	m
	from	**culture/Man_Made_Object** m,
		m/**Person** p

The rewriting algorithm will consider mapping P_3 for a translation but not mapping P_1 since the prefix relationship does not hold for these translations. Recall that translations accepted are those where the prefix relationship between abstract and concrete paths is retained. Hence, the algorithm misses answers.

Furthermore, the algorithm considers a single concrete path for the translation of an abstract query path. This choice makes the complexity of the algorithm linear, nevertheless the algorithm looses again answers. To conclude, we have to note here that the Xyleme rewriting algorithm considers only *full rewritings*, i.e. if there is no translation for an abstract path in a concrete DTD, then the latter is rejected from subsequent translations. We could consider that some cases of incomplete rewritings are taken care by the query relaxation phase.

2.4.3 Infomaster

Infomaster [81] is an information integration system which provides integrated and uniform access to multiple *distributed, heterogeneous, structured* sources, accessible on the Internet.

Global Schema and Source Descriptions Infomaster uses a three-level abstraction hierarchy for modeling the global schema, and the sources. Users formulate their queries in terms of *interface relations*. Data available in a source is also seen as a set of relations called *site relations*. Between the former and the latter, Infomaster defines a set of *base relations*.

Interface and site relations are described in terms of *base relations*. Site relations are described as *views* over the base relations following the *local as view* approach. Interface relations are described as *views* on the base relations following the *global as view* approach. Both views are described in Infomaster as Datalog rules.

Whereas in Information Manifold and most of the data integration projects following the LAV approach, a source contains *always part* of the information, in Infomaster one is able to specify whether a source contains *part* or *all* of the available information (i.e. whether a source is *complete* or *incomplete*).

User Queries and Query Rewriting User queries are formulated in terms of the *interface* relations. Query rewriting is a two-step process. In the first step (called *reduction*), the user query is rewritten in terms of the base relations. This rewriting is done by replacing the interface relations by their definitions. The resulting query is expressed in terms of the base relations only. In the second phase (called *abduction*) the query is rewritten using the *inverse rules* algorithm for queries expressed in terms of *site relations*. The resulting query is an executable *query plan* which only refers to site relations. This is the traditional problem of answering queries using views.

The idea behind the rewriting algorithm is to construct a set of rules that *invert* the view definitions of the sources : i.e. these rules show how to compute tuples for the base relations from tuples in the source relations.

Example Consider the base relations CREATES(NAME,TITLE) and STYLE(TITLE, STYLE). The first associates an artist (attribute NAME) with the artifact (attribute TITLE) she has created. The second associates an artifact (attribute TITLE) with its style (attribute STYLE). Consider a source s_1 which contains the relation ARTIST(NAME,STYLE). The relation stores the name of an artist and the style of her artifacts. The description of site relation s_1 is :

$$ARTIST(name, style) \subseteq CREATES(title, name), STYLE(title, style)$$

The inverse rules algorithm works as follows : for every conjunct in the body of the view (i.e. base relation), an inverse rule is defined. The inverse rules for the previous source description are illustrated below.

$$\begin{array}{lll} CREATES(f_1(name, X), name) & :- & ARTIST(name, X) \\ STYLE(f_1(Y, style), style) & :- & ARTIST(Y, style) \end{array}$$

Intuitively the inverse rules have the following meaning : a tuple of the form $(name, style)$ in the extension of relation ARTIST, is a *witness* of the tuples in the relations CREATES and STYLE :

1. relation CREATES contains a tuple of the form $(T, name)$ for some value of T;
2. the relation STYLE contains a tuple of the form $(T, style)$ for the *same value* of T.

In the example above the value of T is calculated using the functional term $f_1()$ (which is a non-interpreted Skolem function) which takes as input a tuple $(name, style)$ of the site relation ARTIST. There might be several values of T in the source that cause the tuple $(name, style)$ to be in the join of CREATES and STYLE, but at least one such value must exist. In general, for every existential variable that appears in the view definitions, one functional term is created, which is used in the heads of the inverse rules.

Consider for example a query Q which looks for *"the names of artists who have created an artifact whose style is 'Oil on Canvas' "* :

$$Q(name) : -CREATES(title, name), STYLE(title, 'Oil\ on\ Canvas')$$

Suppose that the tuples contained in the site relation ARTIST(NAME,STYLE) are :

$$\{('Van\ Gogh', 'Oil\ on\ Canvas'), ('GeorgesBraque', 'Water\ on\ Canvas')\}$$

The inverse rules compute the following tuples for the base relations CREATES and STYLE :

$$\begin{array}{ll} \text{CREATES:} & \{(f_1('Van\ Gogh', 'Oil\ on\ Canvas'), 'Van\ Gogh') \\ & (f_1('Georges\ Braque', 'Water\ on\ Canvas'), 'Georges\ Braque') \\ \text{STYLE:} & \{(f_1('Van\ Gogh', 'Oil\ on\ Canvas'), 'Oil\ on\ Canvas'), \\ & (f_1('Georges\ Braque', 'Water\ on\ Canvas'), 'Water\ on\ Canvas') \end{array}$$

When the query is evaluated against these tuples, the result is *'Van Gogh'*. The rewriting of a query Q using a set of views V is the Datalog program that includes the inverse rules for V and the query Q.

[80] demonstrates that the inverse rules algorithm returns the *maximally contained rewritings* of Q using the set of views, in time that is polynomial in the size of the query and the views. This algorithm is modular in the sense that the inverse rules can be computed independently for any query (at compile time). The inverse rules algorithm has common points with the bucket algorithm of Information Manifold described previously : each inverse rule can be seen as a kind of bucket which is computed independently of the query. The bucket algorithm for each query subgoal (which is in fact a relation in the global schema) calculates the views that provide answers for it. In a similar manner, the inverse rules algorithm, at compile time, finds for each global schema relation, the views that provide answers for it.
The difference is that the bucket algorithm considers the *conditions* in the query for constructing the buckets, whereas the inverse rules algorithm does not. On the other hand, the query rewriting obtained by the inverse rules may contain views which are not relevant to the user query. Nevertheless, inverse rules are computed only once.

But the inverse rules algorithm has a major drawback : the inverse rules compute the extensions of the mediated schema relations, and then a user query is evaluated over these extensions. In this case, the query re-computes the joins which have already been computed by the view. Consequently the advantage of exploiting the materialized view is lost.

2.4.4 PICSEL

PICSEL [97] relies on a formalism combining the expressive power of rules and classes for designing a rich global schema, thus enabling a fine-grained description of the contents of the sources. The Description Logic language CARIN [137] is used to model both the domain of application and the contents of information sources which are available and relevant to the domain of interest.

The CARIN Language In general Description Logic (DL) languages have been especially designed for modeling and reasoning on complex data descriptions, and their expressive power is well-suited for a natural conceptual modeling of the domain and of the sources. They vary according to the constructors they allow for defining complex concepts and roles. They are associated with inference algorithms that automatically structure the set of concepts, based on the existing subsumption relations between pairs of concepts. When DL's are viewed as a query language, concept subsumption corresponds to query containment.

The CARIN language used in PICSEL, deals with *concepts* (unary relations) that represent sets of objects, and *roles* (binary relations) which represent relationships between objects. Statements in a CARIN knowledge base are definitions of concepts or inclusion statements between concepts and roles. Relations not appearing in CARIN statements are called *ordinary* relations and can be of any arity.

Concept definitions and concept inclusions are formulated using the constructors $\sqcap$ (e.g. $C_1 \sqcap C_2$, where C_1, C_2 are concepts), value restriction ($\forall\ R.C$, where R is a role and C is a concept), number restrictions ($(\geq\ n\ R)$, $(\leq\ n\ R)$) and negation ($\neg\ A$, A is an ordinary concept).
The concept inclusions allowed in PICSEL are of the form $C \sqsubseteq$ *Concept Expression*, where C is an atomic concept and $C_1 \sqcap C_2 \sqsubseteq \bot$, where C_1, C_2 are atomic concepts. A concept is called *atomic* if it does not appear in the left hand side of a definition.

CARIN knowledge base includes also rules which are logical implications of the form : $\forall\ \bar{X}[p_1(\bar{X_1}) \wedge \ldots \wedge p_n(\bar{X_n}) \Rightarrow q(\bar{Y})]$, where $\bar{X_i}$ is a tuple of variables or constants included in $\bar{X}$. Such expressions must be safe : a variable that appears in $\bar{Y}$ must also appear in $\bar{X_1} \cup \ldots \cup \bar{X_n}$. The p_i's are concept names or expressions, or role names and q must be an ordinary relation. Relations that do not

appear in the antecedent of some rule are called *base* relations.

PICSEL Global Schema Given a vocabulary of concept names, the PICSEL global schema is a set of *complex relations* defined using rules or concept expressions as presented earlier.

For example, if the vocabulary contains concepts *Artist*, *Painting*, and role *Paints*, then the new concept *Painter* can be defined as follows[10] :

$$Painter := Artist \sqcap (\geq 5\ Paints) \sqcap (\forall\ Paints.Painting)$$

Rules define new n-ary relations. For example, the rule shown below defines the relation $Cultural\_Artifact$ which contains, the *name* of the artist, the *title* of the object and its *style*.

$$\begin{array}{l} Artist(A) \wedge Painting(P) \wedge \\ Name(A, name) \wedge Paints(A, P) \wedge Title(P, title) \wedge \\ Style(P, style) \Rightarrow Cultural\_Artifact(name, title, style) \end{array}$$

Modeling the sources Each source is characterized by a set of *source relations* (or views). These contain :

- a set of rules of the form $u(\bar{X}) \Rightarrow p(\bar{X})$, where p is a base relation. One such rule is defined for each source relation, following the *local as view* approach. This rule states the kind of data that can be found in the source.
- a set of constraints on the instances of the source relations. These constraints may state inclusions between source relations and concept expressions, or integrity constraints on source relations (for example, negation).

For example, suppose that a source s_1 contains Dutch artists. A second source s_2 contains European painters in general. The views written for the sources are illustrated below.

Sources	Rules	Constraints
s_1	$s_1^1(X) \Rightarrow Artist(X)$ $s_1^2(X,Y) \Rightarrow Born(X,Y)$	$s_1^1 \sqsubseteq (\forall\ Born.Holland)$
s_2	$s_2^1(X) \Rightarrow Painter(X)$ $s_2^2(X,Y) \Rightarrow Born(X,Y)$	$s_2^1 \sqsubseteq (\forall\ Born.Europe)$

Source s_1 returns artists and their place of birth. This is stated in the definitions of the source relations s_1^1 and s_1^2. The constraint involving source relation s_1^1 states that all artists in the source are born in Holland. Source s_2 returns painters and their place of birth. This is stated by the source relations s_2^1 and s_2^2. The constraint on the source relation s_2^1 states that all painters are born in Europe. For the concepts the following hold : $Holland \sqsubseteq Europe$ and $Painter \sqsubseteq Artist$.

Query Processing User queries are conjunctive queries expressed in terms of base relations (concept expressions or role names). For example the user query $Q(X) : -Painter(X) \wedge Born(X,Y) \wedge Holland(Y)$ looks for painters born in Holland. PICSEL rewrites queries by finding conjunctions of source relations (i.e. *rewritings*) which together with the base relations and the source descriptions entail the initial query. The result of the rewriting process in PICSEL is a union of conjunctive queries.

Query $Q_1 : -s_1^1(X) \wedge s_2^1(X) \wedge s_2^2(X,Y)$ is a rewriting of Q since :

[10] This definition states that, a painter is an artist who has painted more that 5 paintings.

s_1^1 entails $(Artist \sqcap \forall\, Born.Holland)(X)$
s_2^1 entails $(Painter \sqcap \forall\, Born.Europe)(X)$

The conjunction $s_1^1(X) \wedge s_2^1(X)$ entails $Painter(X) \wedge \forall\, Born.Holland(X)$. Given the semantics of the $\forall$ universal quantifier, the conjunction of $s_1^1(X) \wedge s_2^1(X) \wedge s_2^2(X,Y)$ entails that there exists Y such that $Painter(X) \wedge Born(X,Y) \wedge Holland(Y)$ is true. Hence, Q_1 is a rewriting of Q.

Query processing in PICSEL is done in the following steps :

- first the query is normalized (for example an expression of the form $C(X) \wedge C'(X)$ is rewritten to $(C \sqcap C')(X)$;
- second satisfiability is tested (of the definition of source relations and the concept expression in the query);
- third, query unfolding is done by replacing the concept expression in the query by the source relations that use it.

2.4.5 Agora

The basic purpose of the Agora system [149, 150, 151] is to support the querying and the integration of heterogeneous *relational* and *XML* information sources. The global schema in Agora is an XML DTD. Relational and XML resources are described as views on this global schema. Users formulate XQuery queries [44] in terms of the global DTD.

Global Schema : The Agora global schema is an XML DTD. Nevertheless, a *virtual, generic, relational* schema is used as an interface between the sources and this schema. It defines for each XML type (element, attribute, processing instruction, comment etc.), a relational table. For example, tables **Element**, **Attribute** are defined to represent XML *element* and *attribute* constructs.

This schema is generic, since it is constructed as a fully normalized version of the hierarchical structure of an XML document. It is virtual, since the actual data resides in the sources and it is never materialized.

The advantage drawn by such a generic schema is that since the generic relational schema is very close to the structure of the XML document, query translation from XQuery expressions to SQL queries is straightforward and is independent of the XML-to-relational mapping.

Source Descriptions and Query Rewriting : Relational and XML resources are modeled as *SQL queries* over the generic relational schema.

The basic idea behind those local-to-global schema mappings, is the following : each relation in a relational data source is considered as an XML document. Each attribute of the relation is modeled as an XML element, child of the root node of the XML document.

Similar to the relational source, an XML resource is also modeled by an SQL query formulated in terms of the generic relational model. In order to exploit a data source stored as an XML file, a DOM wrapper exports relational tables, described as views over the virtual generic relational schema (just like tables from relational data sources).

User Queries and Query Rewriting : The user queries the underlying sources using XQuery expressions formulated in terms of the XML DTD. These are translated into SQL queries expressed in terms of the generic relational schema. The steps before query rewriting are the following :

1. first the query is normalized [149] and brought to a form which is best suited for query translation;

2. given this normalized form, the XQuery expression is translated into an SQL query over the generic relational schema;

3. in the last step, the SQL query over the generic relational schema is rewritten into SQL queries using the relational views.

The rewritten queries are then forwarded to the LeSelect[11] data integration engine which is responsible for the optimization and the distributed execution of the SQL query.

The Agora system does not define new algorithms for the rewriting of SQL queries using relational views. As common in the local as view approach, Agora returns *maximally contained rewritings*. Even if an XML DTD is used as the global schema, an extended use of the relational model is done and query rewriting is based on existing methods and algorithms for rewriting queries using SQL views. The basic contribution is the introduction of rewriting rules for the normalization of XQuery expressions before their translation to SQL queries.

2.5 Data Integration Systems following the Global as View approach

In this section we present the Tsimmis system [46], one of the first to follow the global as view approach to data integration. The MIX [22], Nimble [78] and e-XMLMedia [95] systems are described in Section 2.6. Another system that follows the *global as view* approach is the MOMIS [25] system where a very expressive Description Logic language (ODL-I3) derived from ODMG is used for describing the schemas of the sources to integrate. In MOMIS the global schema is constructed by integrating the local sources schemas.

2.5.1 Tsimmis

The Tsimmis [46] system is one of the first data integration system based on the mediator-wrapper architecture. A simple view of the system architecture is shown in Figure 2.10.

OEM (Object Exchange Model) [225] (a *lightweight* object model) is used to convey information between the components of the system. The mediator is specified using *MSL* (Mediator Specification Language) [171]. It is a *logic-based*, *object oriented* language that can be seen as a view definition language, targeted to the OEM data model. Wrappers are specified using *WSL* (Wrapper Specification Language). It is an extension of MSL, supporting the description of source contents and source query capabilities.

End user queries are written in *LOREL* [5], an extension of OQL [53] targeted to *semi-structured* data. LOREL queries are translated to MSL queries, which are forwarded to the mediator. In the following we present in short the OEM data model, and give an example of MSL specification rules.

OEM : Data represented in OEM is *self-describing* meaning that it can be parsed without any reference to an external schema. It is possible to see OEM as "object oriented" in the sense that the fundamental concept of the model is an *object*. In contrast to object oriented models, the type system of OEM is elementary : an OEM object has an *object-id*, a *label*, a *type* and last a *value*.

[11] LeSelect query execution engine :`http://www-caravel.inria.fr/LeSelect`

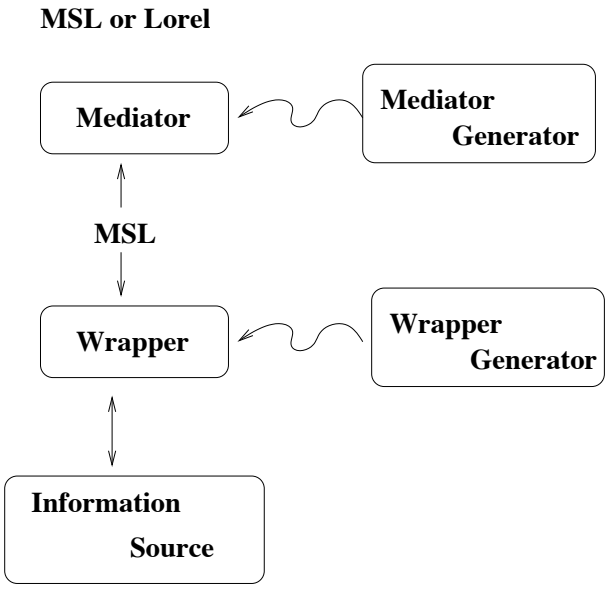

Figure 2.10: Tsimmis Components

The *object-id* may be constructed by the mediators or by the source, to be an expression describing where the object comes from. Unlike object-oriented identifiers, an object's id may be (i) optional, (ii) local to a query, and (iii) not necessarily persistent. *Label* tells what the object represents (in a rough way, its 'class'). Nevertheless, objects with the same label may have different subobjects, or even different types. The *type* of an object can either be an atomic type such as integer, string, or set of OEM objects. Last, its *value* can be an atomic value if the object is atomic, or a set of OEM objects. An OEM object is a triple $<$ *oid label value* $>$. For example the triple

```
<456 cultural_artifact { <title 'Soleil Levant'>}
```

is an OEM object whose object identifier is `456`, its label is `cultural_artifact` and its value is the set { `<title 'Soleil Levant'>` }. `<title 'Soleil Levant'>` is an OEM object whose label is `title` and value is `'Soleil Levant'`. This object is not associated with an object identifier.

Mediator Specification : A mediator is specified using *MSL* (Mediator Specification Language) [171]. Let us illustrate by an example the specification of a mediator in Tsimmis. Consider a mediator that exports information for cultural artifacts. The mediator is specified by the following two rules :

```
(R1)<art(ID) cultural_artifact { <title T>}>@m :-
                    <artifact { <artifact_id ID> <title T>}>@s1

(R2)<art(ID) cultural_artifact { <year Y>}>@m :-
                    <art_object { <art_object_id ID> <year Y>}>@s2
```

This mediator exports objects with label `cultural_artifact`. The `cultural_artifact` objects fuse the objects exported from sources `s1` and `s2` and have the same identifier.

The left hand side part of each of the above rules defines a *view* exported by the mediator. Each rule consists of a head, followed by a `:-` and a tail. The head describes the structure of the mediator

object, and the tail describes the conditions (patterns) that must be satisfied by the source objects. The objects exported by the sources and by the mediator are OEM objects.

The first rule states that :

1. if there is a pair of bindings *id* and *t* for variables `ID` and `T` such that source `s1` contains a `artifact` object that has a `title` sub-object with value *t* and a `artifact_id` sub-object with value *id*,
2. then the mediator exports a `cultural_artifact` object with object-id `art(`*id*`)`, which has a `title` sub-object with value *t*.

For the second rule, the same holds. Notice that none of these rules forbids the addition of new sub-objects to the `cultural_artifact` objects specified by other rules.
MSL allows rules to *incrementally* and *independently* insert information to already exported objects. The semi-structured nature of OEM allows this incremental specification, which would not be the case if there existed some rigid schema (as is the case of Information Manifold and Infomaster systems).

Query Evaluation : User queries are expressed in Lorel [5] query language. They are transformed to MSL queries which are then forwarded to the mediator. Let us illustrate by a simple example how query evaluation is done in Tsimmis. Suppose that a user looks for *"cultural artifacts whose* `ID` *is 456"*. In this case, the MSL query is illustrated below.

```
(Q1)<art('456') cultural_artifact V> :- <art('456') cultural_artifact V>@m
```

The object pattern that appears in the query tail is matched with the head of the mediator specification rule. After this matching, the tails of the mediator specification rule are evaluated against the head of the wrapper specification rules. For query `Q1`, the tails of queries `Q2` and `Q3`, illustrated below are sent to the wrappers of sources `s1` and `s2` respectively.

```
(Q2)<art('456') cultural_artifact { <title T>}> :-
                    <artifact { <artifact_id '456'> <title T>}>@s1

(Q3)<art(ID) cultural_artifact { <year Y>}> :-
                    <art_object { <art_object_id '456'> <year Y>}>@s2
```

Wrappers translate these queries to queries evaluated by the sources. When the answers are obtained, they are transformed using the wrapper specification rules to OEM objects. These are forwarded to the mediator. Finally, two answer objects obtained from different sources are fused into a single `cultural_artifact` object if these share the same identifier.

Semantic keys : An important assumption in the Tsimmis system for performing object fusion is that sources to be integrated have a common way to identify their objects (for example sources s_1 and s_2 identify their `artifact` and `art_object` objects by means of the sub-object `ID`). A number of interesting issues associated with object fusion have been identified in [170] :

Fusion with Canonical Forms : Keys may be represented differently in the sources. This is the problem of *different naming conventions* presented earlier, common to all data integration scenarios. To resolve this problem, data must be *normalized* using a *controlled vocabulary* or a *lexicon* that provides synonyms or equivalences between values. A similar problem concerns

the structure of data. For example one source might represent the name of a person as the pair *(first name, last name)* while a second source might return a simple string for the person's name. In this case, more complex solutions must be used. For example, one can define a function that creates a single value for the name of a person out of the two values in the pair *(first name, last name)*.

Fusion in the absence of keys: In this case, sources might not use keys to identify their objects. To decide, for example, if two person records represent the same real person, one needs to apply a complex function that compares their names, addresses, year and place of birth. The output, is not a canonical key, but mostly a record that combines the available information. The problem here is that fusion can have an unbounded number of steps. This depends on the nature of the values to be compared (if they are complex, or simple).

Another case is that sources might also identify differently their objects. For example, a source might identify a person by her name, and the second by her social security number. This case, can be considered as more specific than the first one, and therefore a similar solution can be applied.

2.6 Bibliographic Notes

MIX MIX (*Mediation of Information using XML*) [22] is a successor of Tsimmis. The difference with Tsimmis is that XML (instead of OEM) is used as the language to represent (i) the global schema and (ii) to exchange data between the mediator and the (XML) sources. More specifically, instance and schema information is represented entirely as XML documents and XML DTDs respectively. The global schema is specified manually by the engineer in charge and describes how the different sources are integrated to the MIX mediator, following the global as view approach.

The query language of MIX is XMAS (*XML Matching And Structuring Language*). This language uses features from several XML query languages such as XML-QL [71], Yat [49], MSL [225] and UnQL [33]. XMAS allows object fusion and pattern matching on XML data. Additionally, XMAS features powerful grouping and order constructs for generating new integrated XML objects out of the existing ones. XMAS queries are formulated in terms of the mediator schema, and are *rewritten* into XMAS queries that refer to the source views exported by the wrappers. The XMAS queries are then sent to the wrapped sources for evaluation.

Nimble Nimble [78] is a commercial system that follows the global as view approach for the integration of XML sources. Similar to the MIX system, Nimble is based entirely on XML. In addition to relational data, data from hierarchical stores and data in structured files can also be handled by the system using appropriate wrappers. The architecture of the Nimble system is based on a set of *mediated schemas*, which are defined as *views* over the schemas of the data sources. Similar to Tsimmis, the mediated schemas can be built in a hierarchical fashion, that is, a mediated schema can be defined on another mediated schema or on an exported source (local) schema. The query language used by the Nimble system is XML-QL [71]. When a query is posed to the integration engine, it is broken into multiple source queries based on the target data sources. The compiler, translates each such query into the appropriate query language for the destination source (for example SQL queries for a relational source).

e-XMLMedia Data Integration Suite [95, 201] is a product which supports (i) the integration of heterogeneous data sources under an XML schema and (ii) the efficient querying of the integrated

information using XQuery [44]. e-XMLMedia supports the integration and querying of sources such as *legacy*, *virtual semi-structured* and finally *loosely structured* ones. It uses XML as the integration model and data exchange format between the different system components and processes XQuery queries in real time by supporting maximum delegation of subqueries to the sources. The two basic components of the system is the *Repository* and the *Mediator*.

The e-XMLMedia Repository stores XML documents in object-relational DBMS and provides XQuery access to retrieve the XML document fragments. The storage of documents is done by *mapping* the XML structure to a collection of relational tables. There are two kinds of mappings : the *generic* and the *specific* schema-based. In the former the tables are automatically generated whilst in the latter mapping directives should be given by the source administrator. The Repository is also able to store XML documents which have no schema through the automatic creation of appropriate *metadata*, similar to data guides. The querying of the repository is done by means of XQuery expressions.

The e-XMLMedia Mediator offers a uniform view and query access of heterogeneous information sources. The mediator (and the wrappers) handle *collections* of XML documents, where each collection is identified by a *name*. A collection originates from a wrapped data source or from even a relation in a relational data source in the repository. When such a collection is 'published' to the mediator, appropriate *metadata* are automatically generated. These record the paths in the XML documents of the collection. The user queries these collections by formulating XQuery expressions in terms of an XML schema which offers a unique view of the underlying data. The schema is the union of all the schemas and the paths that exist in the collections. Query processing consists of *decomposing* the user query (expressed in terms of the XML schema) into mono-source XQuery queries. Each of them is expressed in terms of the structure of a single collection. The decomposition is done using the metadata available for the collections. These are used to verify that a path present in a user query exists in a collection and to type the query expressions. When each of the decomposed queries is evaluated against the source, the results are returned in the form of DOM trees of objects.

The mediator is responsible for (i) the localization of relevant sources, (ii) the amount of sequential and parallel execution using the binding relationships expressed by the query join predicates, (iii) the amount of query processing offloaded onto the data source and (iv) the structuring of the results in an integrated XML document according to the **RETURN** clause of the initial XQuery.

SIMS SIMS [47] was one of the first data integration systems. SIMS considered the integration of *structured* sources and used the LOOM [145] Description Logics system to manage the *global schema*, and the source descriptions.

The sources are integrated, by specializing existing global schema classes (LOOM classes) and queries are defined in terms of the latter. Query rewriting is performed using *reformulation operators* of the LOOM system. Given a query, the system uses the source descriptions to first select the databases that possibly answer the query. This is done by using the LOOM operator `select database()`. If there is no database that provides answers to the query, then the query is reformulated into one using more general concepts by the operator `generalize concept()`. Last, the attributes requested in the original query are examined. If for example the query requests all actors (instances of concept Actor) and their paintings (instances of concept Painting), then this is reformulated into one requesting *painters* instead of actors. This is due to the fact that the role *paints* used in the query is defined in concept Painter and not in concept Actor. Another operator is `partition concept()` which considers set coverings for 'breaking' the query into sets of queries. Consider for example a query which requests instances of concept Actor. If concepts Painter and

Sculptor are subconcepts of Actor and are disjoint, then a query requesting actors will break down to a union of two queries, one requesting painters and the other sculptors. Given the generated queries, the LOOM system considers the evaluation of the queries as a *planning problem.*

Ontology-Based Data Integration Systems *OBSERVER* [154] is one of the first systems to use *ontologies* for data integration. The basic assumption in OBSERVER is that each resource is described by an *ontology.* These ontologies are linked explicitly by the *inter-ontology* relationships (synonymy, hypernymy, and hyponymy). By these relationships information in the repositories is linked as well. These ontologies can be considered as *source descriptions.* In OBSERVER, query evaluation is performed as follows : a user selects an ontology and expresses her query in terms of it. The system then accesses the underlying sources described by the selected ontology to answer the user query. If the user is not satisfied by the answers, then the system selects a different ontology. In order to translate the user query into the newly selected ontology, the two ontologies (user and target ontology) are integrated by taking under consideration the inter-ontology relationships between them. This integration takes place in a CLASSIC [28] knowledge base. An apparent problem that arises here is that if ontologies are large, and if they contain overlapping concepts then integration is not a straightforward process. Authors assume that ontologies and inter-ontology relationships are small.

Carnot [58] uses the knowledge base CYC [134] for describing source contents. CYC knowledge base is a formalized representation of a *"vast quantity of fundamental human knowledge"* and contains about 10^5 general concepts and 10^6 assertions on these concepts. An information source is integrated in Carnot by providing mapping rules between the source structures and CYC structures in the form of *articulation axioms.* User queries are formulated using the CYC structures, which are then translated into local structures through the established mapping rules.

OntoSeek [106] supports content-based access to the Web, designed for content based information retrieval from on-line yellow pages and product catalogs. OntoSeek combines an ontology driven content matching facility with a moderately expressive representation formalism. In OntoSeek the ontology incorporated is Sensus [126] which is based on WordNet [161]. Resources are encoded as *linguistic conceptual graphs*, and OntoSeek allows the use of arbitrary expressions for this process, resolved using the previously mentioned ontologies. Similarly to resource descriptions, *queries* are also encoded as graphs and the problem of query answering is reduced to *graph matching.* The bottleneck of this approach is that it permits semantically not valid expressions, although they are perfectly linguistically valid. Moreover, Sensus contents do not often correspond to real world relationships between classes of entities in the world making difficult the precise encoding of information. OntoSeek also states the difficulty introduced by the absence of defining explicitly that concepts of the Sensus ontology are disjoint. For this reason, authors state that it is important to introduce a top level ontology that will have a structuring role over the Sensus ontology.

Chapter 3

Generation of Metadata Schemas

In this chapter we demonstrate our approach for the *creation of metadata schemas* by the integration of *pre-existing ontologies* and *thesauri*.

Section 3.1 gives a formal definition of the notions of *thesaurus* and *ontology* and Section 3.2 describes our approach for the construction of metadata schemas by the integration of these structures.

In Section 3.3 we illustrate two applications for the constructed metadata schemas. The first is the *creation of RDF [179] schemas* (Section 3.3.1). The second is the *definition* and *creation* of *source content descriptive metadata* (Section 3.3.2). Section 3.4 presents two object oriented implementations of a metadata repository. Our approach was validated by a National project between the CNAM (Conservatoire National des Arts et Métiers) and the French Ministry of Culture and the result of this collaboration is the ELIOT prototype presented in Section 3.5.

3.1 Ontologies and Thesauri

3.1.1 Thesauri

A thesaurus is a set of controlled *terms* organized with the fixed set of *associative*, *equivalence* and *hierarchical* relationships. Hierarchical relationships include the *generic* (*btg*), *whole-part* (*btp*) and *instance* (*bt*) relationships. Most of the thesauri use the *btg* relationship which is transitive, carries subset semantics and is the most frequent relationship in monolingual thesauri. In our work we consider thesauri constructed with the *btg* relationship which organizes terms related to a specific subject into directed acyclic graphs, referred to as *hierarchies*. Two examples of *btg*-hierarchies are shown in Figures 3.1 and 3.2. For example, in Figure 3.1, term *paintings* is a broader term of *oil paintings*, with the interpretation that all objects that belong to the extension of the latter, belong also to the extension of the former. A hierarchy is defined by its *root term*, a term with no broader term (e.g. <*visual works*> in Figure 3.1). We only assume mono-hierarchical thesauri, i.e. each term has exactly one broader term.

In the following, a thesaurus is considered as a *forest* of *thesaurus hierarchies*. Although the following definition of thesauri is not complete w.r.t. all possible term relations of existing thesauri, it is sufficient for creating rich metadata schemas.

Definition 3.1.1 *A* thesaurus hierarchy H *is a rooted tree,* $H = (D, E)$ *where :*

- $D = \{t_0, t_1, t_2, \ldots t_n\}$ *is the set of descriptors in* H *(nodes of the tree);* t_0 *is a distinguished descriptor, called the* root *of* H*;*

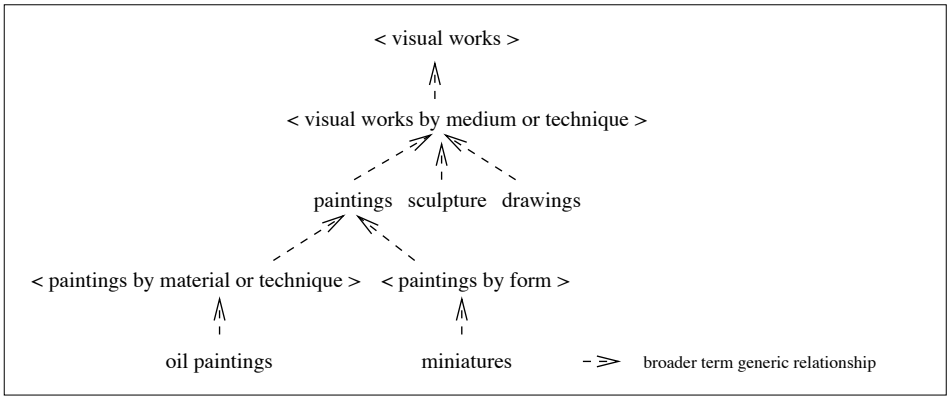

Figure 3.1: Part of the Art & Architecture Thesaurus hierarchy *Visual Works* which collects all artifacts that are used for visual communication (paintings, sculptures, photos).

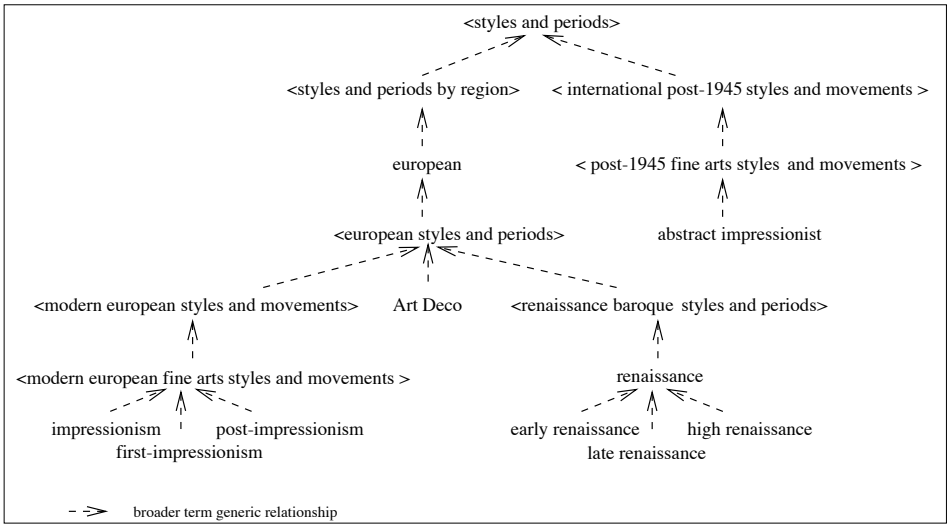

Figure 3.2: Part of the Art & Architecture Thesaurus hierarchy *Styles & Periods* which collects all styles, periods and movements of Art in the western world.

- *E is a binary relation on D (i.e. $E \subset D \times D$).*

If $(t,t') \in E$, then (t,t') is an *edge* of H. Edges denote the *btg* relation between descriptors. An edge (t,t') defines that t is related by a *btg* relationship to t'. In this case t is the *broader term* of t' in the hierarchy and t' is called the *narrower* term of t.

A sequence of edges $(t_1,t_2),(t_2,t_3)\ldots(t_{n-1},t_n)$ is called a *btg-path* from t_1 to t_n. For all terms t_i,t_j, $j > i$, if there exists a *btg*-path from t_i to t_j, then all the instances in the extension of t_j are in the extension of t_i. *btg* defines a *partial order* between terms in a hierarchy : it is reflexive, antisymmetric and transitive.

Definition 3.1.2 *A* thesaurus *is a forest of* thesaurus hierarchies. *Let $\{H_1,H_2,\ldots H_n\}$ be the set of hierarchies, where $H_i = (D_i,E_i)$ and $\forall\ i,j,\ i \neq j$ then $D_i \cap D_j = \emptyset$. Then a thesaurus $\mathcal{T}$, is a pair $\mathcal{T} = (\mathcal{D},\mathcal{E})$ where :*

- *$\mathcal{D}$ is the set of descriptors in $\mathcal{T}$: $\mathcal{D} = \bigcup_{1\leq i\leq n} D_i$;*
- *$\mathcal{E}$ is the set of edges in $\mathcal{T}$: $\mathcal{E} = \bigcup_{1\leq i\leq n} E_i$;*

3.1.2 Ontologies

Ontologies are defined independently of the actual data [103], reflect a common understanding of the semantics of the domain of discourse and are used to share and exchange semantic information between sources and users [102]. They are declarative specifications of the basic concepts and roles in an application domain. In our work we consider ontologies with inheritance relations (*isa*) and typed roles, sufficient to describe a large class of domains [105].

Definition 3.1.3 *An ontology is a 5-tuple $\mathcal{O} = (C,V,R,A,isa)$ defined as follows :*

1. *$C = \{c_1,c_2,\ldots,c_n\}$ is a set of concepts, where each concept c_i refers to a set of real world entities (concept instances),*
2. *V is a set of atomic types (*Integer, String, *etc.),*
3. *$R = \{r_1,r_2,\ldots,r_m\}$ is a set of binary typed roles between concepts,*
4. *$A = \{a_1,a_2,\ldots a_k\}$ is a set of attributes defined between concepts in C and atomic types in V, and*
5. *isa is an inheritance relationship defined between concepts. It carries subset semantics and defines a partial order over concepts.*

Ontologies are represented as directed, labeled graphs where nodes correspond to concepts and atomic types, and arcs correspond to roles, attributes and *isa* relationships. Figure 3.3 illustrates an example ontology, inspired from the ICOM/CIDOC Reference Model [112] which is used to describe cultural information. The ontology describes concepts such as Man_Made_Object, its subconcept Iconographic_Object, Actor and its subconcept Person. The ontology describes actors (persons, organizations, instances of concept Actor), activities (instances of concept Activity) and man made objects (instances of concept Man_Made_Object). The fact that some actor carries out some activity is represented by the role *carried_out* between concept Actor and concept Activity. A man made object has a title, represented by the attribute *has_title* of type String. Role *of-period* defined in concept Man_Made_Object describes that an object belongs to some period (like

renaissance, moyen age). Concept Iconographic_Object, subconcept of Man_Made_Object, inherits the roles and attributes defined in the latter. The fact that iconographic objects have a style, is represented by the role *style* defined between concepts Iconographic_Object and Style. For each role $r(c, d)$ defined between concepts c and d, its *inverse* denoted by $r^{-}(d, c)$ is defined. For our example ontology, the inverse of role *carried_out* is *carried_out_by*. Inverse roles are depicted in the figure within parentheses.

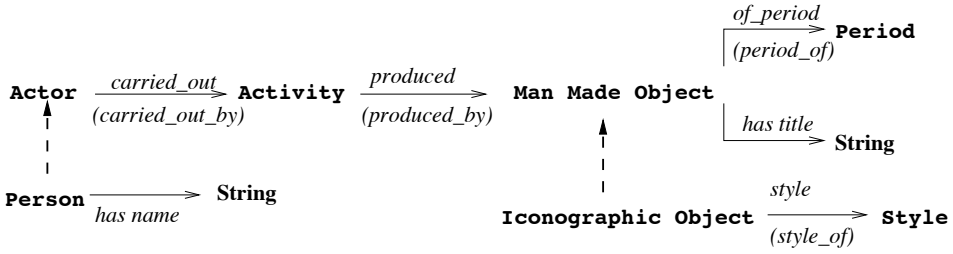

Figure 3.3: A simple cultural ontology.

3.2 Creating Metadata Schemas from Ontologies and Thesauri

In this section, we present our methodology for the construction of metadata schemas by the integration of *existing ontologies* and *thesauri*.

Recall that ontologies represent the basic notions of the domain of interest by means of *concepts*, *roles* and *attributes*. Thesauri consist of hierarchies of terms which are constructed using *a priori* defined semantic relationships which are restricted to hierarchical, equivalence and associative.

Let us consider the *Art & Architecture (AAT)* [1] thesaurus which organizes terms in *facets*, i.e. general categories of interest in the art and architecture domains. Although this thesaurus contains extended hierarchies of terms, it does not contain semantic relationships (other than the related term relationship presented in Section 2.1.2) which allow to navigate from one hierarchy to another. Consider for example the record for descriptor *paintings* shown in Figure 3.4. The descriptor belongs to the hierarchy *Visual Works*. The scope note gives additional information about the term. Remark that descriptors *paintings* and *painting (image-making)* are connected by the *related term* relationship. The latter belongs in the *Processes and Techniques Hierarchy* of *AAT*. The *related term* relationship is used here to associate *artifacts* with the *processes* used for their creation.

Consider the ULAN (Union List of Artist Names) [203] thesaurus which is an example of a thesaurus adding semantic information in the form of *natural text* comments. Figure 3.5 shows the record which contains information about the painter *Dossi Dosso*.

The information described in the record cannot be represented using the fixed set of semantic relationships (i.e. hierarchical, associative and equivalence) encoded in a thesaurus. For example, consider entry `life dates` (line 5) in the record which contains information about the dates of death and birth of the painter (i.e. the dates associated to these *events*). Moreover, the short comment next to the painter's name (line 2) gives additional information about the *place* where these events happened. The problem here is that since this data is recorded as full text and not represented explicitly (for example by means of *relationships*), precise structured queries cannot be expressed.

```
Descriptor: paintings

Term ID: 33618

Hierarchy: Visual Works [VC]

Scope note - Use for unique works in which images are formed
primarily by the direct application of pigments suspended in oil,
water, egg yolk, molten wax, or other liquid, arranged in masses of
color, onto a generally two-dimensional surface.

Synonyms and spelling variants {UF}: pictures

Related concepts {RT} : painting (image-making)
```

Figure 3.4: The record for descriptor *paintings* of the AAT

```
1.  Record ID : 9329
2.  Dossi Dosso (Italian Painter;born in Ferrara 1490;died in Ferrara 1541-1542)
3.  note : Although early biographies, including Vasari, noted a birth
           date of 1475, modern scholars agree that Dosso cannot have
           been .....

4.  Names :
    Dossi Dosso (prefered)
    De Lutero, Giovanni,
    Dossi di Ferrara
    ....

5.  life dates:
    born 1490, active from 1512, died 1542

6.  roles:
    painter, draftsman

7.  geographical locations
    Ferrara (Italy)
    Venice (italy)

8.  related people
    student of : Lorenzo Costa di Ottavio, from 1507
```

Figure 3.5: A record of the ULAN thesaurus

Consider now the entry `related people` (line 8). This information is recorded in the ULAN thesaurus by the *associative* relationship (*rt*). Nevertheless, one can see that the thesaurus developers have specified that Dosso was the *student of* Lorenzo Costa di Ottavio and not his teacher for example. The same relationship is used to describe that a person *belongs to* a group of people who worked together to produce an artifact, to record the *role* of a person, and finally to represent her *membership* in the group.

The above examples demonstrate that thesauri record semantic relationships between terms either using the *related term* relationship or by introducing natural text comments in term records. On the other hand, observe the example ontology presented in Figure 3.3. This ontology contains concepts that describe the basic notions in the domain of interest (e.g. Man_Made_Object and Style), but does not contain more specific notions (e.g. *paintings* or *impressionism*). But, it contains *rich semantic* relationships between the concepts.
To conclude, we can make two observations :

- the first is that with thesauri, one can express at a *fine granularity* level the semantics of the source contents using precise terms. But the relationships between these terms are fixed, and richer semantic relationships do not exist.

- the second is that ontologies allow one to create structured source descriptions. But, the level of detail of the ontology as far as concept hierarchies are concerned, is shallow compared to that of thesauri.

For thesauri, the user can express queries using very precise terms, but one cannot express *structured* queries. For example she can express queries such as *"painting and Van Gogh"* but not queries such as *"painting created by Van Gogh"*. The latter query is more specific than the former since it determines explicitly the relationship between terms *"painting"* and *"Van Gogh"*.

To conclude, the idea is to produce metadata schemas that incorporate the two views of information :

- *ontologies* that specify rich *semantic relationships* in the domain of discourse and

- *thesauri* which incorporate *precise semantics* specified in term hierarchies.

The construction of a metadata schema is done in three steps :

1. In a first step, one specifies for each ontology concept, a set of terms, considered as its *subconcepts*. This step is similar to establishing *inter-schema assertions* [39, 58] for database schema integration. The result of this step is a *connection relation*.

2. In a second intermediate step, for each concept, a *concept thesaurus* is *automatically* extracted. This thesaurus contains only the terms connected to this concept by the connection relation, along with *btg* (*broader-generic*) relationships derived from the initial thesaurus. This process can be done automatically and does not require the knowledge of the ontology.

3. In the final step these thesauri are *integrated* with the ontology to produce a metadata schema consisting of (1) a *structural view* provided by the ontology, (2) *connection relations* between concepts and terms, and (3) *thesaurus hierarchies*.

Our approach can be applied for the construction of metadata schemas *independently* of the final formalism used to represent them.

Step 1 : Connecting Terms to Concepts In this step, thesaurus terms are "connected" to ontology concepts. These connections have inclusion semantics and are represented by a binary *connection relation* $Con \subseteq \mathcal{D} \times C$ where $\mathcal{D}$ is the set of descriptors in thesaurus $\mathcal{T}$, and C is a set of ontology concepts in ontology $\mathcal{O}$.

An example of a connection relation is presented in Figure 3.6. Terms *impressionism*, *post-impressionism* and *abstract impressionism* of the *Art & Architecture Thesaurus* hierarchy *Styles & Periods* (Figure 3.2) describe specific styles (ontology concept Style in Figure 3.3). Term *first-impressionism* of the same hierarchy describes both a style and a period (concepts Style and Period respectively). Similarly, term *renaissance* and its narrower terms in the *Styles & Periods* hierarchy describe different types of styles and periods (ontology concepts Style and Period respectively). Finally, terms *paintings*, *oil paintings* and *sculpture* of the *Art & Architecture* thesaurus hierarchy *Visual Works* (Figure 3.1) define different kinds of iconographic objects (ontology concept Iconographic Object).

Term	Concept
impressionism	Style
post-impressionism	Style
abstract impressionism	Style
renaissance	Style
renaissance	Period
late renaissance	Style
late renaissance	Period
first-impressionism	Period

Term	Concept
paintings	Iconographic Object
oil paintings	Iconographic Object
sculpture	Iconographic Object
early renaissance	Style
early renaissance	Period
high renaissance	Style
high renaissance	Period
first-impressionism	Style

Figure 3.6: A connection relation *Con* for AAT hierarchies *Styles & Periods*, *Visual Works* and ontology concepts Style, Period and Iconographic Object.

In the previous example, we do not connect the whole thesaurus hierarchy *Styles & Periods* to concepts Style and Period. This *selective approach*, i.e. relating thesaurus terms to ontology concepts explicitly, is chosen for several reasons. An obvious reason is that some terms could be out of the scope of the application that has to be described by the resulting metadata schema. For example, if some application is only concerned with paintings, then terms referring to artifacts other than paintings (e.g. *sculpture*, *drawings*) need not be considered in the resulting schema. Another reason is that some terms (e.g. *guide terms* in [114, 194]) are used to organize thesaurus hierarchies (e.g. term <*visual works by medium or technique*> in Figure 3.1) and might have no use in describing information. Finally, another important reason is that thesaurus hierarchies might contain terms which can be connected to different concepts. For example, terms of the *Art & Architecture* thesaurus hierarchy *Styles & Periods* (Figure 3.2) describe styles (e.g. *impressionism*), periods (e.g. *art deco*), or both styles and periods (e.g. *renaissance*). Connecting terms to concepts in a selective manner allows users to clarify between the multiple semantics of a term (e.g. as in the case of *homonyms*) and consequently resolve semantic ambiguities at the thesaurus level.

Step 2 : Thesaurus Extraction After having defined the connection relation between terms and concepts, a thesaurus, called *concept thesaurus* is extracted for each concept present in this relation. This is done in two steps :

- First, a labeled thesaurus denoted by $\mathcal{T}_\lambda$ is created out of the connection relation $Con \subseteq \mathcal{D} \times C$ and thesaurus $\mathcal{T}$. Each term in $\mathcal{T}_\lambda$ is labeled with the set of concepts to which it is connected

in relation *Con*. Observe that a term can be connected to several concepts. For example, in the connection relation illustrated in Figure 3.6, term *first-impressionism* is connected to both Style and Period concepts. In this case, the label of term *first-impressionism* is the set of concepts {Style, Period}.

- Second, a selection operation σ is defined, which constructs from a labeled thesaurus $\mathcal{T}_\lambda$ and a set of concept names $S \subseteq C$, a new labeled thesaurus which contains (1) the set of terms in $\mathcal{T}_\lambda$ whose labels contain at least one concept in S and (2) *btg* relations between these terms, induced by the *btg* relations in the initial thesaurus.

Creation of the labeled thesaurus $\mathcal{T}_\lambda$

Definition 3.2.1 *Let $\mathcal{T} = (\mathcal{D}, \mathcal{E})$ be a thesaurus, and $\mathcal{O} = (C, V, R, A, isa)$ be an ontology. Let $Con \subseteq \mathcal{D} \times C$ be a connection relation between descriptors in $\mathcal{T}$ and concepts in $\mathcal{O}$. A labeled thesaurus $\mathcal{T}_\lambda$ is a 4-tuple, $\mathcal{T}_\lambda = (\mathcal{D}, \mathcal{E}, \lambda, \mathcal{C})$ where :*

- *$\mathcal{D}$ is the set of descriptors in $\mathcal{T}$;*
- *$\mathcal{E}$ is the set of edges in $\mathcal{T}$;*
- *$\mathcal{C}$ is a set of concepts in $\mathcal{O}$ ($\mathcal{C} \subseteq C$);*
- *λ is a labeling function $\lambda : \mathcal{D} \rightarrow 2^{\mathcal{C}}$, such that $\forall t \in \mathcal{D}$ and $c_i \in \mathcal{C}$, $c_i \in \lambda(t) \Leftrightarrow (t, c_i) \in Con$.*

Definition 3.2.2 *Let $\mathcal{T}_\lambda = (\mathcal{D}, \mathcal{E}, \lambda, \mathcal{C})$ be a labeled thesaurus. Let $S \subseteq \mathcal{C}$ be a set of concepts. The selection operation $\sigma(S, \mathcal{T}_\lambda)$ returns a new labeled thesaurus $\mathcal{T}_S = (\mathcal{D}_S, \mathcal{E}_S, \lambda_S, S)$ where :*

1. *$\mathcal{D}_S$ is the set of descriptors, $\mathcal{D}_S \subseteq \mathcal{D}$ where $\forall\ t \in \mathcal{D}_S$, $\lambda(t) \cap S \neq \emptyset$;*
2. *let t, t' be two descriptors in $\mathcal{D}_S$. Edge (t, t') is in $\mathcal{E}_S$ iff there do not exist descriptors $\{t_1, t_2, \ldots t_n\}$ in $\mathcal{D}_S$ where $(t, t_1), (t_2, t_3) \ldots (t_n, t')$ is a btg-path in $\mathcal{T}_\lambda$;*
3. *$\forall\ t \in \mathcal{D}_S$, $\lambda_S(t) = \lambda(t) \cap S$.*

Let us now illustrate the algorithm calculating $\sigma(S, \mathcal{T}_\lambda)$ which is shown in Figure 3.7. The input is (i) a labeled thesaurus $\mathcal{T}_\lambda$ and (ii) the set of concepts S. It returns the thesaurus $\mathcal{T}_S$ i.e. the restriction of the labeled thesaurus $\mathcal{T}_\lambda$ to the set of terms labeled by at least one concept in S. The algorithm navigates in the thesaurus starting from the root terms and then navigating downwards in the hierarchy. To create the thesaurus $\mathcal{T}_S$ it uses the procedure *create_btg* which is presented in Figure 3.8. This procedure (i) keeps the terms that are labeled with some concept in S and (ii) creates the *btg* relationships between these terms, out of the ones in $\mathcal{T}_\lambda$.

The algorithm illustrated in Figure 3.7, calls for each root term r (line 4) and for each of its narrower terms t (line 5) the procedure *create_btg*(r, t) (line 6) which updates $\mathcal{T}_S$. Procedure *create_btg* is illustrated in Figure 3.8 and proceeds as follows : let (t_i, t_j) be the terms with which *create_btg* is called, for which there exists a *btg*-path in $\mathcal{T}_\lambda$ (i.e. t_i is the broader term of t_j in $\mathcal{T}_\lambda$). If t_i is not labeled by some concept in S (line 5) then *create_btg* is called for t_j and its children (lines 6-7). Otherwise, t_i is added in $\mathcal{D}_S$ (line 11). Then the algorithm examines whether t_j is labeled with some concept in S (line 14). If this is the case, then t_j is added in $\mathcal{D}_S$ and the edge (t_i, t_j) is added in $\mathcal{E}_S$ (lines 16,18). *create_btg* is then called for t_j and its children (line 21). Otherwise, i.e. t_j is not labeled with some concept in S (line 24), then *create_btg* is called for t_i and the children of t_j (lines 26, 27).

1. Input : a set of concepts S, a labeled thesaurus $\mathcal{T}_\lambda = (\mathcal{D}, \mathcal{E}, \lambda, \mathcal{C})$
2. Output : a thesaurus $\mathcal{T}_S = (\mathcal{D}_S, \mathcal{E}_S, \lambda_S, S)$
3. Algorithm : Initialization : $\mathcal{D}_S = \emptyset$, $\mathcal{E}_S = \emptyset$
4. for all root terms r in $\mathcal{D}$ {
5. for all terms t such that $(r,t) \in \mathcal{E}$ {
6. $\mathcal{T}_S = create\_btg(r, t, \mathcal{T}_\lambda)$
7. }
8. }
9. return $\mathcal{T}_S$.

Figure 3.7: Algorithm which calculates $\sigma(S, \mathcal{T}_\lambda)$

1. Input: Terms t_i, t_j (t_i broader term of t_j), labeled thesaurus $\mathcal{T}_\lambda$
2. Output : the thesaurus $\mathcal{T}_S$
3. Algorithm : */* if t_i is not labeled by some concept in S, */*
4. */* call the procedure for the children of t_j */*
5. if $\lambda(t_i) \cap S = \emptyset$ {
6. for all terms t such that $(t_j, t) \in \mathcal{E}$
7. $create\_btg(t_j, t, \mathcal{T}_\lambda)$
8. */* if t_i is labeled by some concept in S */*
9. } else {
10. */* add t_i to $\mathcal{T}_S$ */*
11. $\mathcal{D}_S = \mathcal{D}_S \cup \{t_i\}$, $\lambda_S(t_i) = \lambda(t) \cap S$
12. */* examine t_j */*
13. */* if t_j is labeled by some concept in S */*
14. if $\lambda(t_j) \cap S \neq \emptyset$ {
15. */* add t_j to $\mathcal{T}_S$ */*
16. $\mathcal{D}_S = \mathcal{D}_S \cup \{t_j\}$, $\lambda_S(t_j) = \lambda(t_j) \cap S$
17. */* create btg relation between t_i and t_j */*
18. $\mathcal{E}_S = \mathcal{E}_S \cup \{(t_i, t_j)\}$
19. */* call the procedure for the children of t_j */*
20. for all terms t such that $(t_j, t) \in \mathcal{E}$
21. $create\_btg(t_j, t, \mathcal{T}_\lambda)$
22. } else {
23. */* if t_j is not labeled by some concept in S */*
24. if $\lambda(t_j) \cap S = \emptyset$ {
25. */* call the procedure for the children of t_j */*
26. for all terms t such that $(t_j, t) \in \mathcal{E}$
27. $create\_btg(t_i, t, \mathcal{T}_\lambda)$
28.
29. }
30. }
31. return $\mathcal{T}_S$.

Figure 3.8: Procedure *create_btg*

Defining the concept thesaurus : Using the selection operation σ it is possible to define a thesaurus $\mathcal{T}_c$ for each concept c in the set of ontology concepts C.

Definition 3.2.3 *Let $\mathcal{T}_\lambda = (\mathcal{D}, \mathcal{E}, \lambda, \mathcal{C})$ be a labeled thesaurus. Let $\mathcal{O} = (C, V, R, A, isa)$ be an ontology, where $\mathcal{C} \subseteq C$. Let c be a concept in $\mathcal{C}$ and S_c be the set of subconcepts of c in $\mathcal{O}$ including c. The* concept thesaurus *$\mathcal{T}_c = \sigma(S_c, \mathcal{T}_\lambda)$ contains all terms $t \in \mathcal{D}$ such that $\lambda(t) \cap S_c \neq \emptyset$.*

For the definition of a concept thesaurus, *btg* relations between terms *and isa* relationships at the ontology level are exploited. Consider the example in Figure 3.9. Term v is labeled by concept d, term t by c and w by e. The selection operation on concept c constructs the thesaurus $\mathcal{T}_c$ that contains besides term t, terms v and w that are labeled by c's sub-concepts.
Observe that a term can appear in multiple concept thesauri, and terms that are not labeled by any concept have disappeared from the concept thesauri. For example, term u is not connected to any concept and has disappeared from the concept thesauri in Figure 3.9. Moreover, the selection operation on concept c created a *btg* relation between terms t and w which were not directly related by a *btg* relationship in the original thesaurus.

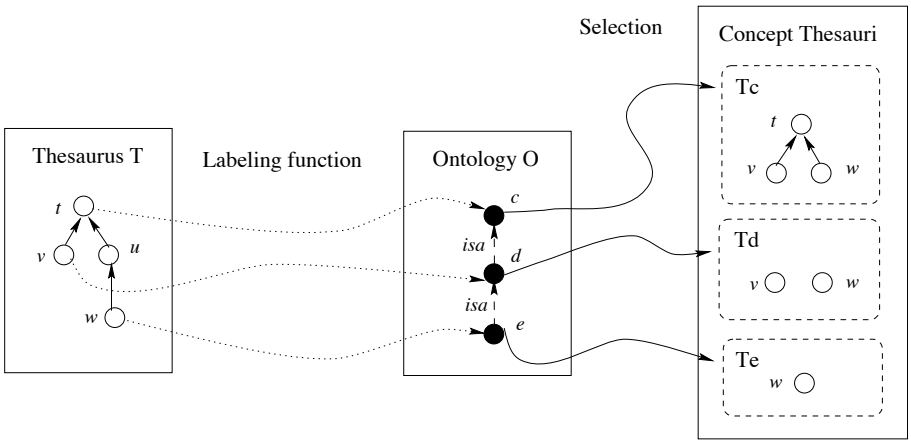

Figure 3.9: Extracted Thesaurus Examples

A concept thesaurus can be induced by those of its superconcepts : if d is a sub-concept of c, and $\mathcal{T}_c$ is the concept thesaurus of c, then the concept thesaurus of d can be extracted as follows : $\mathcal{T}_d = \sigma(S_d, \mathcal{T}_c)$. Concept thesauri generally represent only a small part of the original thesaurus. In the previous example, only concept thesaurus $\mathcal{T}_c$ of concept c has to be extracted from the original thesaurus $\mathcal{T}_\lambda$. Then, the concept thesauri of its subconcepts (i.e. d and e) can be extracted from $\mathcal{T}_c$, by using the selection operation σ.

Figure 3.10 shows the concept thesauri of concepts Iconographic_Object, Style and Period constructed out of the connection relation of Figure 3.6 and the *Art & Architecture* thesaurus hierarchies shown in Figures 3.1 and 3.2.

Step 3 : Creation of the metadata schema At this final step, the metadata schema is created by integrating the ontology with the concept thesauri extracted in the previous step. To create the metadata schema, the *isa* relationships between concepts and the *btg* relationships between terms are considered. Relationships *btg* and *isa* define a partial order on terms and concepts respectively and have inclusion semantics.

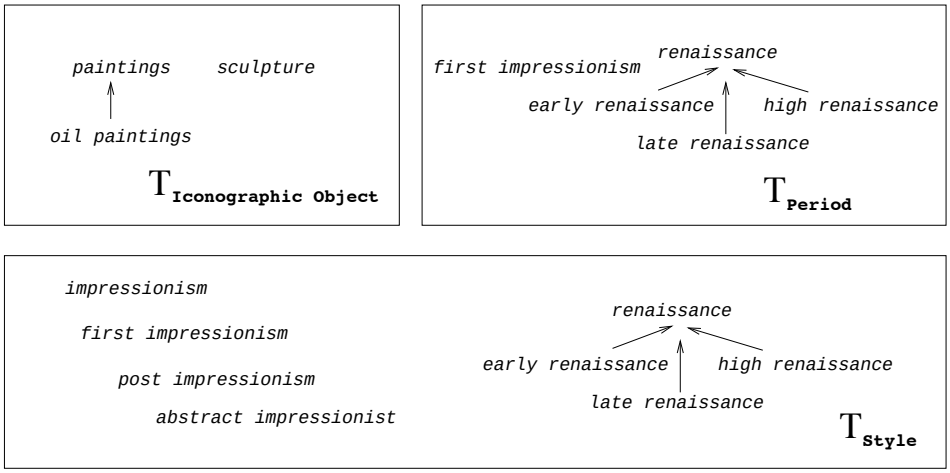

Figure 3.10: Concept Thesauri for ontology concepts Iconographic_Object, Style and Period

Let us now give a more formal definition of the metadata schema $\mathcal{M}$ resulting from the integration of the initial ontology $\mathcal{O}$ and a set of concept thesauri.

Definition 3.2.4 *Let $\mathcal{O} = (C, V, R, A, isa)$ be an ontology. Let $T = \{\mathcal{T}_{c_1}, \mathcal{T}_{c_2}, \ldots \mathcal{T}_{c_n}\}$ be a set of concept thesauri where $\mathcal{T}_{c_i} = (\mathcal{D}_{c_i}, \mathcal{E}_{c_i})$ is the concept thesaurus of concept c_i. The metadata schema $\mathcal{M}$ is $\mathcal{M} = (C_\mathcal{M}, V, R, A, isa_\mathcal{M})$ where :*

- *$C_\mathcal{M}$ is the set of concepts in $\mathcal{O}$ and the terms in all $\mathcal{T}_{c_i}$: $C_\mathcal{M} = C \cup \{c_i : t \mid t \in \mathcal{D}_{c_i}\}$;*
- *V is the set of atomic types, R is the set of roles and A is the set of attributes in $\mathcal{O}$;*
- *$isa_\mathcal{M}$ is obtained by the isa relations between the concepts in ontology $\mathcal{O}$. It is extended by the following isa relations :*
  1. *for each concept c_i create an $isa(c_i : t, c_i)$ relation to all concepts $c_i : t$ where $c_i : t$ corresponds to the root term t in thesaurus $\mathcal{T}_{c_i}$.*
  2. *for edge $(t, t') \in \mathcal{E}_{c_i}$, create an isa relation $isa(c_i : t', c_i : t)$ in $\mathcal{M}$.*

Example 3.2.1 *Figure 3.11 illustrates a part of the metadata schema constructed from the ontology in Figure 3.3 and the concept thesauri shown in Figure 3.10. For term* renaissance *which belongs to the concept thesauri of both concepts* Style *and* Period*, the* concepts Style:renaissance *and* Period:renaissance *are introduced to distinguish between the notion of "renaissance" as a style and as a period. Concepts* Style:renaissance *and* Period:renaissance *become sub-concepts of concepts* Style *and* Period *respectively. Term* oil painting *is a narrower term of* paintings *in the concept thesaurus of* Iconographic Object*. Concepts* Iconographic Object:oil painting *and* Iconographic Object:paintings *are created for these terms where the former becomes a subconcept of the latter. The latter is a subconcept of concept* Iconographic Object*.*

Let us now examine in more detail the resulting metadata schema. The concept thesauri of two concepts c, c' where c *isa* c' are *overlapping* : the concept thesaurus for c' contains the terms that

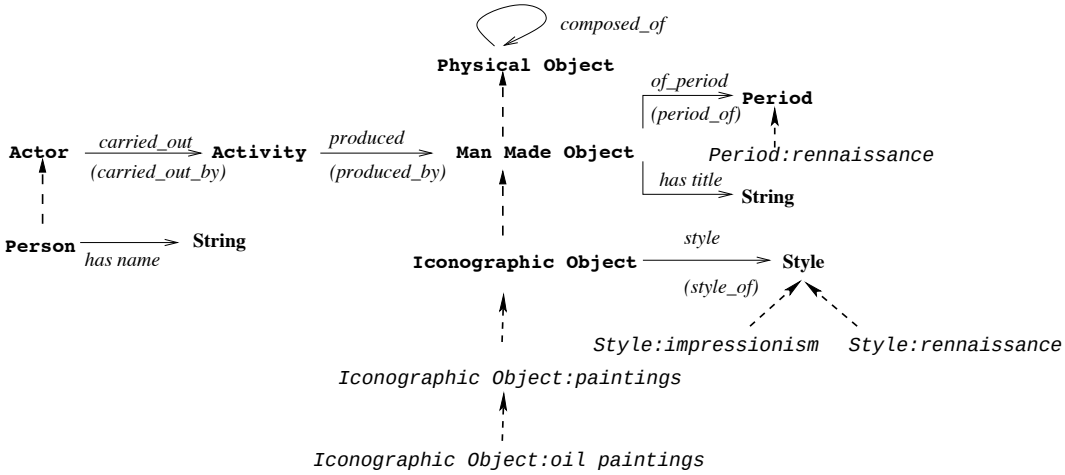

Figure 3.11: Example of a metadata schema

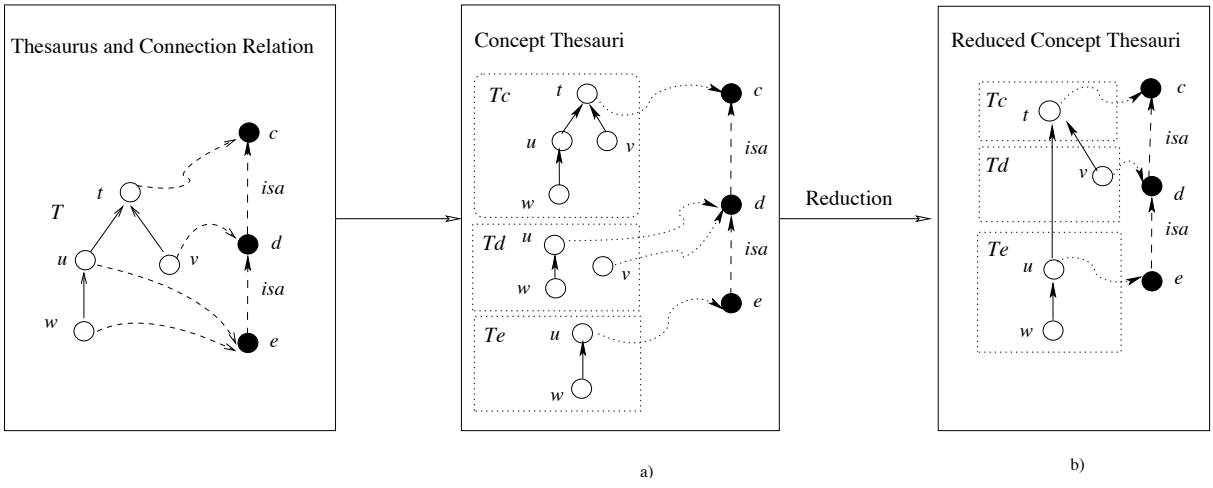

Figure 3.12: Reducing the Number of Terms

are directly connected to c' in the connection relation *and* the terms that belong to the concept thesaurus of its subconcept c.

For example, the thesauri extracted for concepts c, d and e in Figure 3.12-(a) contain 9 terms. Based on the fact that *btg* and *isa* have inclusion semantics one can consider the set of these relations as a directed acyclic graph defined over terms and concepts where the existence of a path between some term or concept a and some other term or concept b signifies that a is included in b. If for example t is a term in the thesaurus $\mathcal{T}_c$ and is the root of a hierarchy in a thesaurus $\mathcal{T}_d$ where d is a sub-concept of c, then the same subtree in the first thesaurus can be replaced by the same subtree in the thesaurus $\mathcal{T}_d$. An example for this kind of reduction is shown in Figure 3.12. After reduction, thesaurus $\mathcal{T}_c$ only contains term t. The subtrees below t have been replaced by *btg inter-thesaurus* relations from thesaurus $\mathcal{T}_d$ and $\mathcal{T}_e$. A similar reduction was possible in thesaurus $\mathcal{T}_d$. Observe that it is possible to create all original paths connecting terms and concepts in the reduced representation of the original DAG defined by *btg* and *isa*.

3.3 RDF Metadata Schemas and Resource Descriptions

In this section two applications for the metadata schemas produced following our approach, are presented. The first is the creation of RDF [179] schemas (Section 3.3.1) and the second is the *creation* and *querying* of metadata descriptions (Section 3.3.2).

3.3.1 Controlled Creation of RDF Schemas

This section illustrates how an RDF schema can be constructed out of a set of *concept thesauri*. This schema incorporates the set of *ontology concepts*, *roles* and *attributes*, the *concept thesauri* defined for each ontology concept and *connections* between terms and concepts. More precisely :

- ontology concepts and thesaurus terms are modeled as RDF *classes*;
- ontology roles and attributes are modeled as RDF *properties*;
- ontology *isa* relationships and *btg* relations between terms all carry inclusion semantics and are modeled with the RDF *subclassOf* property.

The creation of the RDF schema $\mathcal{S}$ for an ontology $\mathcal{O} = (C, V, R, A, isa)$, and a set of concept thesauri $\mathcal{T}_c = (\mathcal{D}_c, \mathcal{E}_c)$ is straightforward :

1. The set of RDF *classes* in $\mathcal{S}$ is obtained as follows:
   (a) for each ontology concept c define RDF class c;
   (b) for each term t in $\mathcal{T}_c$ define RDF class c:t.
2. The set of RDF properties is obtained by defining :
   (a) for each typed role $r(c,d)$ in R, where c and d are concepts in C, an RDF property with domain RDF class c and range RDF class d;
   (b) for each typed attribute $a(c,v)$ in A where c is a concept in C and v is an atomic type in V, an RDF property with domain RDF class c and range a literal.
3. The set of RDF *subclassOf* properties is obtained as follows:

(a) for each $isa(c, d)$ relationship between ontology concepts c and d, define an RDF subclassOf property between RDF classes c and d;

(b) for each RDF class c:t, corresponding to a root term in thesaurus $\mathcal{T}_c$, add an RDF subclassOf property between RDF classes c:t and c;

(c) for each *btg* relation between two terms t and t' (t is the broader term of t') in a concept thesaurus $\mathcal{T}_c$, define an RDF subclassOf property between RDF classes c:t' and c:t.

It is interesting to note that only the RDF class corresponding to a root term in the concept thesaurus of concept c is connected to RDF class c. Due to the transitivity of the RDF *subclassOf* property, it can be induced that class c:t is a subclass of class c:t' or of class c.

The RDF schema illustrated in Figure 3.13 has been constructed from the ontology in Figure 3.3 and the concept thesauri in Figure 3.10.

Ontology concepts Man Made Object, Iconographic Object, Style, Period and terms *paintings*, *oil paintings*, *impressionism* and *first-impressionism* are all represented as RDF classes (lines 5,-7,9,10,21,23,25,27). For simplification, terms are prefixed with the corresponding concept if they are contained in different concept thesauri. RDF class paintings is defined as a subclass of class Iconographic Object (line 22), since term *paintings* is the root term of the Iconographic Object concept thesaurus. In the same way, classes impressionism and first-impressionism are defined as subclasses of concepts Style and Period respectively (lines 26,28). Class oil paintings is a subclass of class paintings (line 24) (defined by the *btg*-relation between terms *paintings* and *oil paintings*). Ontology role *style*, is defined as an RDF property, its domain being the class Iconographic Object (line 19) and its range, class Style (line 20). By the definition of the RDF subclassOf property, all subclasses of class Iconographic Object inherit this property.

RDF Descriptions Using this RDF schema, one can provide RDF descriptions about specific Web resources. Consider for example the RDF description presented in Figure 2.2. The new RDF description is shown in Figure 3.14. When comparing this new description with the previous one, one can observe that the prefix artifact is replaced by a new prefix int which corresponds to the RDF schema in Figure 3.13. In this RDF description, semantic information that was captured as a value in the previous description has been added at the schema level. For example, the fact that resource `http://metalab.unc.edu/louvre/paint/monet/first/impression/` described an impressionist painting was encoded in the value of tag <artifact:style>. This value corresponds in fact to a term in the *Art & Architecture* thesaurus and is represented as an instance of class int:impressionism (line 11) in the new description. The same argument holds for the value *first-impressionism* which is now represented as an instance of RDF class int:first-impressionism (line 12). Observe also that the tag <rdf:Description> (Figure 2.2, line 7) has been replaced by a typed node tag <int:oil paintings> (line 8) indicating that the resource is about an oil painting.

3.3.2 Metadata Descriptions

In this section we present a simple approach for creating metadata [19] using the metadata schema produced with the methodology described in Section 3.2.

A source can be either a collection of documents, a single document or even a document fragment. It is identified by a *url* and is integrated in the portal by providing a *description* of its structure, contents or even of some semantics which is not explicit in the data or in its structure. This description is created in terms of the *metadata schema*. At that time, the source is said to be

```
1. <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2.          xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
3.          xmlns:int="">
4.   <rdfs:Class rdf:ID="Physical Object"></rdfs:Class>
5.   <rdfs:Class rdf:ID="Man-Made Object">
6.     <rdfs:subclassOf rdf:resource="#Physical Object"/></rdfs:Class>
7.   <rdfs:Class rdf:ID="Iconographic Object">
8.     <rdfs:subclassOf rdf:resource="#Man-Made Object"/></rdfs:Class>
9.   <rdfs:Class rdf:ID="Period"></rdfs:Class>
10.  <rdfs:Class rdf:ID="Style"></rdfs:Class>
11.  <rdf:Property rdf:ID="of-period">
12.    <rdfs:domain rdf:resource="#Man Made Object"/>
13.    <rdfs:range  rdf:resource="#Period"/></rdf:Property>
14.  <rdf:Property rdf:ID="has_title">
15.    <rdfs:domain rdf:resource="#Man Made Object"/>
16.    <rdfs:range rdf:resource="http://www.w3.org/2000/03/example/classes/#String"/>
17   </rdf:Property>
18.  <rdf:Property rdf:ID="style">
19.    <rdfs:domain rdf:resource="#Iconographic Object"/>
20.    <rdfs:range  rdf:resource="#Style"/></rdf:Property>
21.  <rdfs:Class rdf:ID="paintings">
22.    <rdfs:subclassOf rdf:resource="#Iconographic Object"/></rdfs:Class>
23.  <rdfs:Class rdf:ID="oil paintings">
24.    <rdfs:subclassOf rdf:resource="#paintings"/></rdfs:Class>
25.  <rdfs:Class rdf:ID="impressionism">
26.    <rdfs:subclassOf rdf:resource="#Style"/></rdfs:Class>
27.  <rdfs:Class rdf:ID="first-impressionism">
28.    <rdfs:subclassOf rdf:resource="#Period"/></rdfs:Class>
29. </rdf:RDF>
```

Figure 3.13: The RDF schema resulting from the integration of the ontology and thesaurus.

```
1. <rdf:RDF
2.  xmlns:web-page ="http://metalab.unc.edu/louvre/namespaces/web-pages"
3.  xmlns:int ="http://www.connectit.com/icom/aat">
4.  <rdf:Description
5.   about="http://metalab.unc.edu/louvre/paint/monet/first/highway/">
6.   <web-page:title>Web Museum: Monet, Claude :Impression :
                                        soleil levant</web-page:title>
7.    <web-page:presents>
8.     <int:oil paintings
9.      about="http://metalab.unc.edu/louvre/paintings/monet/impression">
10.     <int:title>Impression : soleil levant</int:title>
11.      <int:style><int:impressionism/></int:style>
12.      <int:of-period><int:first-impressionism/>
13       </int:of-period>
14.   </int:oil paintings>
15.   </web-page:presents>
16.   <web-page:creator>Nicolas Pioch</web-page:creator>
17.  </rdf:Description>
18.</rdf:RDF>
```

Figure 3.14: RDF description for Claude Monet painting using the integrated schema.

published and can be exploited for querying. The pair *(schema, set of source descriptions)* is called the *description base* (DB).

A user can query the description base by formulating queries in terms of the metadata schema. The result to these requests is a set of metadata descriptions. In contrast to data integration systems, a Web portal does not exploit the actual data for answering these queries. The query evaluation stops at the exploitation of the available source descriptions. In a data integration system, these descriptions can be used for the *resource discovery* phase. During this phase the system is based on the source descriptions to select a number of sources that possibly answer the query.

In this part of the work we restrict our attention to the first issue which is the identification of the relevant sources for answering a user query using their metadata descriptions.

Our approach views DB as a *database* queried for *source addresses* (*urls*). Once the user has obtained a set of urls, she is able to access each of them.

In the following, we first define the *resource description model*. An implementation of this model is discussed in Section 3.4.

Resource Description Model We illustrate in the following how a metadata description is defined for a source in terms of the metadata schema constructed in Section 3.2.

A source description is an *instance* of a *concept* in the metadata schema. More formally :

Definition 3.3.1 *Let s be a Web resource with url u. Let $\mathcal{M} = (C_{\mathcal{M}}, V, R, A, isa_{\mathcal{M}})$ be a metadata schema. A source description for s is a tuple (u, c, d) where:*

1. *u is the URL of s;*

2. *c is a concept in $C_{\mathcal{M}}$;*

3. and d is a concept descriptor for concept c.

Definition 3.3.2 *A concept descriptor for concept c is a tuple* $d = (p_1 = d_1, ..., p_n = d_n)$*, where* p_i *is a role or attribute of concept c and* d_i *is a concept descriptor. Let* $P(c)$ *denote the set of role/attributes defined in concept c. The following holds for concept descriptor* d[1] *:*

- $p_i \in P(c)$ *and* $p_i \neq p_j$ *for all* $i \neq j$*, i.e. each role/attribute can be used at most once in a concept descriptor;*
- *if* p_i *is an attribute a, such that* $a(c,t)$*, where t is an atomic type in V, then* d_i *is a value of type t;*
- *if* p_i *is a role r such that* $r(c,c')$ *then* $d_i = (c', d')$ *where* d' *is a concept descriptor for concept* c'*.* d_i *is called an* unnamed *descriptor and is an instance of concept* c'*.*

A *description base* DB is a set of source descriptions. It corresponds to a set of sources published at a given time instant, each of the sources is classified by one or more concepts of the metadata schema. Instances of a concept in the DB are either source descriptions, (designated by a *url*), or *unnamed* descriptors. The DB is the union of the source descriptions, instances of all concepts. A source description, instance of a concept c, is related to unnamed descriptors in DB by the set of roles of concept c. We can make the following remarks :

- if there is no source description associated with concept c, then the extension of c is empty;
- if (u, c, d) is in the extension of c and c' is a *super-concept* of c, then (u, c', d) is in the extension of c', i.e. all resources that are described by a source description (u, c, d) are also described by a source description (u, c', d) (inclusion semantics of *isa*).

Example 3.3.1 *Let us now give some examples of source descriptions which rely on the metadata schema of Figure 3.11.*

*Consider source with url*1 *which is about actors in general. Its description is :*

$$(url1, \mathsf{Actor}, [])$$

It defines no properties on concept Actor *(i.e. use of the empty descriptor* []*).*

*Consider now source with url*2 *which contains data concerning Claude Monet. Its source description is :*

$$(url2, \mathsf{Person}, has\_name = \text{“}ClaudeMonet\text{”})$$

The value of the attribute has_name *is restricted to the value 'Claude Monet' to state that the source contains information about this person.*

*Source with url*3 *is also about Claude Monet. But, this source contains data associated with Monet as a painter. The description of the source is :*

$$(url3, \mathsf{Person : painter}, has\_name = \text{“}ClaudeMonet\text{”})$$

Concept Person:painter *is defined for term* painter *which belongs to the concept thesaurus of concept* Person.

[1] $d = []$ denotes the empty descriptor.

Source with url4 is about activities that are carried out by actors. Its description is :

$$(url4, \mathsf{Activity}, carried\_out\_by = (\mathsf{Actor}, []))$$

Finally consider the last source url5 which is about impressionistic paintings. Its description is shown below

$$(url5, \mathsf{Iconographic\_Object} : \mathsf{paintings}, of\_period = (\mathsf{Period} : \mathsf{impressionism}, []))$$

3.4 Object Oriented Implementation of a DB

In this section a prototype implementation of the DB with the object-oriented database system (ODBMS) O_2 [91] and its querying with the OQL query language [53] are described.

Two object oriented implementations of our metadata schema as far as the representation of thesauri is concerned are given. In both implementations, concepts of the metadata schema are represented as O_2 classes. A role (attribute) defined in some concept is represented as an attribute defined in the concept's respective class. In the first implementation, thesaurus terms are represented as O_2 classes, while in the second, they are represented as values. We argue later the advantages and the disadvantages of the two approaches.

3.4.1 Representing Concepts as Classes

The object oriented implementation of the metadata schema $\mathcal{M} = (C_{\mathcal{M}}, V, R, A, isa_{\mathcal{M}})$ created in Section 3.2 is straightforward. More specifically :

- For each concept c, an O_2 class c is defined[2].
- For each role r, such that r is defined between concepts c and d, an attribute r is defined in class c and takes its values in class d.
- For each attribute a, where a is defined between concept c and an atomic type v, an attribute a between class c and the corresponding atomic type of v in O_2 is defined.
- For each *isa* relationship between concepts c and d, it is specified in the definition of class c that it inherits from class d.

Consider the schema illustrated in Figure 3.11. A part of the O_2 schema defined for concepts Man_Made_Object and Iconographic_Object is illustrated below.

```
class Man_Made_Object
      type tuple( of_period : Period,
                  has_title : string)
end

class Iconographic_Object inherit Man_Made_Object
      type tuple( style : Style)
end
```

[2]In all the examples below, the reserved keywords of O_2 are represented in **this** font. Class names are represented in this font, and finally attributes defined in classes are represented in *this* font.

3.4.2 Representing Thesaurus Terms as Classes

Concepts in the metadata schema corresponding to terms in the thesaurus are represented as O_2 classes. For example the O_2 classes for concepts Iconographic Object:paintings and Iconographic_Object:-oil paintings are defined as follows :

class Iconographic_Object:paintings **inherit** Iconographic_Object **end**

class Iconographic_Object:oil paintings **inherit** Iconographic_Object:paintings **end**

Recall that in the metadata schema concept Iconographic_Object: oil paintings is a subconcept of Iconographic_Object: paintings.

Source Descriptions and Queries For the source descriptions, an O_2 class called Source_-Description is defined. Each class defined for a concept, inherits from this class. The attribute *url* is defined in this class which takes its values in string and stores the url of the source. Finally, the value of an object o, instance of a class c is a *tuple* whose attributes are (i) a mandatory attribute with name *url*, and (ii) at most as many attributes defined in class c. The definition of the O_2 classes Source_Description and Man_Made_Object are illustrated below.

class Source_Description
    type tuple(*url* : string)
end

class Man_Made_Object **inherit** Source_Description
    type tuple(*of_period* : Period,
                has_title : string)
end

For each class c a persistent root cS (database entry point) of type set(c) is defined which contains all objects of class c (all descriptions of concept c).

The formulation of queries in this case is straightforward. For example, if one looks for *"sources about paintings whose title is 'Madonna with Child' "*, one must formulate the query illustrated below. This query will retrieve all descriptions in the persistent root Iconographic Object:paintingsS.

```
select d.url
from d in Iconographic Object:paintingsS
where d.has_title = 'Madonna with Child'
```

3.4.3 Representing Thesaurus Terms as Values

In this implementation thesaurus hierarchies are encoded as *trees* at the instance level. Terms are represented as objects, *instances* of class Term. In this class the attributes *term_name*, *bt* and *nt* are defined. The first stores the name of a term t. The second stores t's broader term and the last stores the set of its narrower terms. The O_2 specification of the class Term is illustrated below.

class Term
    type tuple(*term_name* : string,
                bt : Term
                nt : set(Term))
end

The value of each object o, instance of class Term, is a tuple of type $[n, b, n']$ where n is the name of the term (of type string), b refers to its broader term and n' is the set of its narrower terms.

To represent the thesaurus for a concept c, the persistent root tc of type set(Term) is defined which contains all terms in the concept thesaurus of c.

Source Descriptions In this implementation, the definition of class Source_Description changes and the attribute *term* of type Term is added :

```
class Source_Description
      type tuple( url : string,
                  term : Term)
end
```

As previously, a class c is a subclass of Source_Description whose extension is the set of the extensions of all classes c. The value of an object o, instance of a class c, is a *tuple* whose attributes are (i) a mandatory attribute with name *url* storing the url of the source described by the object, (ii) a mandatory attribute *term* which has for value a thesaurus term and (iii) at most as many attributes defined in class c.

The peculiarity of this representation comes from the semantics associated with the term hierarchies in concept thesauri : if $(u, c, d = [term = t, p_2 = d_2, ..., p_n = d_n])$ is a source description belonging to the extension of class c, then for all broader terms t' in the thesaurus connected to c (that is all terms that belong to the persistent root tc), the source description $(u, c, d' = [term = t', p_2 = d_2, ..., p_n = d_n])$ also belongs to the extension of c. In other words, if (u, c, d) is a source description in DB, then (u, c, d') also belongs to DB.
For example, (i) term *paintings* and *oil_paintings* are terms in the concept thesaurus of Iconographic Object, (ii) term *paintings* is a broader term of *oil_paintings*. If a source u in DB is about *oil_paintings*, then the description base describes this source as being also about *paintings*.

Query Evaluation To evaluate a query over the database, thesaurus traversals are required. For this, the method specific_terms():set(Term) is defined in class Term which returns the set of all narrower terms of the term which calls it. The O_2C code for this method is illustrated below.

```
method body specific_terms:set(Term) in class Term{
/* variables declaration */
o2 Term t1; o2 Term t;
o2 list(Term) lterms;
o2 set(Term) ret = set(self);
/* calling term (self) is added in the result set */

/* run through the narrower terms of calling term (self) */
for (t1 in (*(self->nt)) where t1!=nil){
/* add in lterms the narrower terms of calling term */
  lterms +=list(t1);
}

/* for all terms in lterms call recursively the method specific_terms */
  for (t in lterms){
    ret += set(t) + t->specific_terms;
```

```
  }

/* return the set of all narrower terms */
return ret;
}
```

The query language used to query the DB is OQL. The user specifies a path in the ontology using the attributes of some class. A query always returns a set of urls. A few examples of queries on the schema of Figure 3.11 are illustrated below.

Example 3.4.1

1. *Sources about actors?*

   ```
   select d.url
   from d in Actors
   ```

   This query selects source descriptions, instances of class Actor[3].

2. *Sources about painters ?*

   ```
   select d.url
   from d in Actors, t in tActor
   where t.term_name = 'painter' and
         d.term in t.specific_terms
   ```

   This query selects all source descriptions, instances of class Actor *for which the term used in the description is* painter *or one of its narrower terms*[4].

3. *Sources about activities concerning man-made objects of the renaissance period?*

   ```
   select d.url
   from d in Activities, t in TPeriod
   where t.term_name = 'renaissance' and
         d.produced.of_period.term in t.specific_terms
   ```

   This query selects all activities (instances of class Activity*) that* produced *an object (instance of class* Man-Made Object*) of the* renaissance *period (attribute* of-period *is of type* Period*).*

4. *Sources about Picasso as a sculptor?*

   ```
   select d.url
   from d in Actors, t in tActor
   where d.has_name = 'Picasso' and t.term_name = 'sculptor' and
         d.term in t.specific_terms
   ```

   This query selects all objects, instances of class Actor*, whose name is "Picasso" and the term used to describe the actor is* sculptor *or one of its narrower terms.*

[3] Actors is the persistent root defined in O_2 for concept Actor. It contains all source descriptions, instances of this concept.

[4] tActor is the persistent root defined in O_2 for concept Actor which contains all terms that are in its concept thesaurus.

5. *Sources about painters of sculptures?*

```
select a.carried_out_by.url
from a in Activities, t1 in tActor, t2 in tMan_Made_Object
where t1.term_name = 'painter' and t2.term_name = 'sculpture' and
      a.carried_out_by.term in t1.specific_terms and
      a.produced.term in t2.specific_terms
```

This query selects all sources, instances of class Actor*, (note that attribute* carried_out_by *of class* Activity *is of type* Actor*). The term associated with the actor is* painter *or one of its narrower terms. The term associated with a man-made object (recall that attribute* produced *of class* Activity *is of type* Man Made Object*) is* sculpture *or one of its narrower terms.*

Conclusions We have described previously two ways of implementing thesaurus terms. In the first terms are represented as classes while in the second as values. The problem with the first implementation is that (i) the resulting schema is large (recall that the *Art & Architecture Thesaurus* contains 120.000 terms) and (ii) maintaining the instances is a cumbersome task. To be more precise on this second issue, when a description is added in DB, instance of some class c, then it must be added in the persistent roots of all superclasses of c. We have seen previously that when a description is added for class Iconographic Object:oil paintings, it is considered as a description for Iconographic Object:paintings where the latter is a superconcept of the former. It is evident that the lower a term is in the thesaurus hierarchy, the number of classes in whose persistent roots the description must be added, increases. But, querying is straightforward : thesaurus traversals correspond to traversals of class hierarchies. Moreover, the representation of thesaurus terms as classes is not necessary since *no new roles* or *attributes* are defined on thesaurus terms.

On the other hand, in the second implementation where thesaurus terms are represented as values, the number of schema classes is restricted to the number of concepts in the ontology (a relatively small number). Moreover, the problem of maintenance of the persistent roots of classes in the first implementation is not a problem any more. This is taken care by the method *specific_terms()* used in the OQL queries. But, the presence of this method makes thesaurus traversals non optimized. In the following we discuss this problem and propose a solution where *labeling schemes* are used for encoding thesaurus hierarchies. In this case, thesaurus traversals requested by queries are transformed into interval queries.

Linear encoding of Thesauri The above implementation takes advantage of the efficient optimization of OQL except in the presence of method *specific_terms* in the query's **where** clause. This method considers all the narrower terms of the term present in the original query and requires, in the worst case, a complete traversal of term hierarchies. Moreover, user defined methods cannot be optimized : even if efficient indexes like B-trees or R-trees are used to store the terms, methods do not take advantage of such structures. Consequently, thesaurus traversal is not only costly but may also lead to non-optimal query execution plans. This is particularly true for queries that require the traversal of several large hierarchies of terms. When description bases including thesauri with thousands of terms are queried, these traversals become a central issue.

Labeling schemes are a solution to this two-fold problem :

- optimization of costly thesaurus traversals using standard optimization techniques and
- efficient computation for a number of operations such as *"get all descendants"* of a term.

```
Q : select a.carried_out_by.url
    from a in Activities, t1 in tActor, t2 in tMan_Made_Object
    where t1.term_name = 'painter' and t2.term_name = 'sculpture' and
          a.carried_out_by.term in t1.specific_terms and
          a.produced.term in t2.specific_terms


Q' : select a.carried_out_by.url
     from a in Activities, lb in LActors, lm in LManMadeObjects
     where lb.term = ''painter'' and lm.term = ''sculture'' and
     a.carried_out_by.label between (lb.label, lb.nextlabel) and
     a.produced.label between (lm.label, lm.nextlabel)
```

Figure 3.15: Query Q using the user defined method *specific_terms* and its equivalent Q' which uses the Dewey encoding

More specifically, the idea is to find a *labeling scheme* for terms that takes into account the *btg* hierarchy, and then to map *thesaurus traversal* queries into *equivalent interval queries* on a flat domain. These queries can then be efficiently answered by standard DBMS query languages without the presence of user defined methods. There exist a number of labeling schemes for large hierarchies (see Bibliographic Notes in Section 3.5.3). In the following, we illustrate how thesaurus hierarchies can be efficiently traversed using the Dewey labeling scheme [43] which has been implemented in our prototype [129].

Dewey Encoding The Dewey encoding [43] is one of the simplest labeling schemes. A term hierarchy in this scheme is represented as follows : terms are ranked from left to right with a rank between $[1, max]$ where max is the *fan-out* of the thesaurus. The fan out specifies the maximum number of sons that a term can have.
Let n be a node in the tree. We label n with a string over the integers i, $1 \leq i \leq max$. The string length is d, if n is at depth d in the tree. The i-th character in the label of n is equal to m if the i-th node in the path from root to n has for rank m (it is the mth of its siblings from left to right).

Let $label(t)$ denote the label of term t. Then the ascending lexical order on term labels is a total order with the following property : all labels in the sub-tree with root t are larger than $label(t)$ and smaller than $label(t')$ where t' is the next sibling of t on the right in the thesaurus. As an example (assume for simplicity that max=9), the next sibling of node n with label 22 has for a label 23 and the descendants of n are labeled by 221, 2221, 2222,..., 2227, 222 and 223. Terms labeled with 221, 222 and 223 are sons of node 22. A thesaurus term is represented by a string label and descriptions can be indexed (with the system B-tree) on the term label as any regular O_2 object. Hence, a typical query such as descendants of term t (obtained by the method *specific_terms* in the last implementation), becomes an interval query on node labels : if term n is labeled by 222, then the call of method *specific_terms*() for n becomes the interval query $[222, 223[$.
More generally, let $next(n)$ be the label of the next sibling of n on the right. Query "t in n.*specific_terms*" becomes the interval query "$label(t)$ between $[label(n), label(next(n))[$". Consider for example how the OQL query Q is rewritten into Q' using the Dewey encoding. Both queries are illustrated in Figure 3.15.

In the previous presentation we assume that a node label is an integer. The physical represen-

tation of this integer depends on the value of *max*, i.e. the maximum fan-out of the thesaurus. A naive implementation would take as many bytes per integer as required by *max* (for example the data type INT of 2 bytes if $max \leq 2^{15} - 1$. Then, not only this representation is memory costly if the average fan-out is smaller than *max*, but also a strategy must be designed for the case where after some updates in the hierarchy, the fan-out exceeds *max*. A solution to this problem is to use the widely spread UNICODE format [218] which adapts the 'character' physical storage to its value.

Implementation Given the Dewey encoding scheme, the implementation of the thesaurus hierarchies in an O_2 database is done as follows : For each class c[5], a class Lc is defined containing the labels of the concept thesaurus for c, i.e. defining for each term *t*, its label and its next sibling label. The definition of class LMan_Made_Object that contains the thesaurus terms for class Man_Made_Object is illustrated below.

```
class LMan_Made_Object
      type tuple( term : string,
                  label : string,
                  nextlabel : string)
end
```

Each class Lc is associated with the entry point Lcs which is a collection of objects of class Lc. For example, the range of lb in query Q' above, is the collection (labels) LActors. It corresponds to the term *painter* in the concept thesaurus of Actor and lb.nextlabel is the label of the next sibling of this term in the same thesaurus. Compared to the query Q one can observe that the traversal of the thesaurus using the method *specific_terms* is replaced by two interval queries on labels of the collections LActors and LManMadeObjects.

When encoding schemes for thesaurus terms are used, the descriptions must be preprocessed in order to replace terms by labels. In summary, the advantage of this solution is two-fold : it allows to process tree traversals by standard database optimization techniques on interval queries and the performance gain to be obtained should be significant for queries involving several large thesauri and a number of criteria on thesaurus terms.

The Dewey encoding scheme was used to encode the 120.000 terms of the *Art & Architecture Thesaurus*. The performance of the operation that returns the narrower terms of a term in a hierarchy using the Dewey encoding, was not significantly better than the thesaurus traversals using the user-defined method *specific_terms()*. One of the reasons is that the thesaurus was not large enough. We have not worked further on this subject. Currently authors in [51] study the performance of different labeling schemes on huge description bases such as the Open Directory of Netscape [168].

Multidimensional indexes Last, if the query involves several thesauri traversals, the query can be transformed into a hyper-rectangle query on a multidimensional space of labels. As an example, take the query Q in Figure 3.15 which looks for *"painters of sculptures"*. Any s_1 description having for a term a narrower term of 'sculpture' and any s_2 description having for a term a narrower term of 'painter', is a candidate for the answer. Such a pair is a point in the two dimensional space with coordinates *a.carried_out_by.label* and *a.produced.label*. Then the last two conditions in the **where** clause of query Q' in Figure 3.15, can be replaced by a window query. This is illustrated by Figure 3.16: all points contained in the rectangle represent couples of descriptions (*urls*) on Actor

[5]Recall that a class c is defined for a concept *c* in the metadata schema

and Man-Made Object, are candidates for the query answer. If queries involving both a thesaurus traversal on the concept thesaurus of Actor and Man-Made Object are frequent, it is worth creating a *2-dimensional index* on Actor labels and Man-Made Object labels.

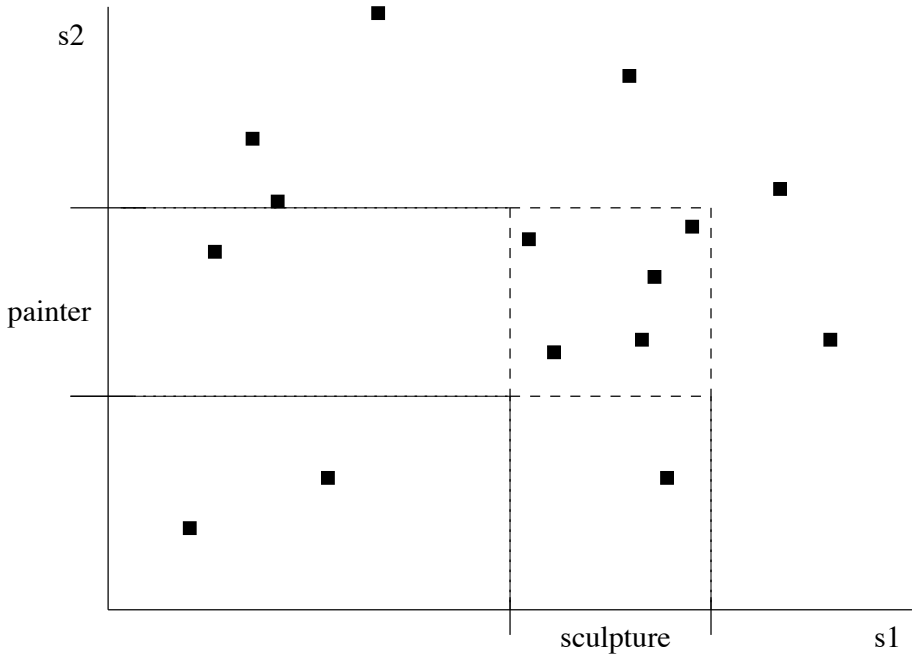

Figure 3.16: Two-dimensional Thesaurus Index

2-dimensional (Spatial) indexing methods [93, 183] are not yet fully integrated in the kernel of off the shelf DBMS. However a current trend of these DBMS is to provide simple and fairly efficient extensions to handle spatial data. As an example, [188] describes the spatial extension of the relational DBMS Oracle 8i.

3.5 The ELIOT Cultural Portal

In this section we describe the prototype ELIOT [176], resulting from a collaboration between the CNAM (Conservatoire National des Arts et Métiers) and the Direction of Heritage and Architecture (Direction de l'Architecture et de Patrimoine -DAPA-) of the French Ministry of Culture.

A number of existing resources such as *thesauri*, *databases*, and indexing tools have been developed by the French Ministry of Culture :

- MERIMEE, which stores approximately 140.000 descriptions of historical monuments,
- PALISSY, stores 200.000 descriptions of mobile objects ranging from religious objects, to domestic ones, scientific and industrial ones,
- ARCHI XX, contains approximately 1000 descriptions and images of furniture of the 20th century, protected under the law of historical monuments,
- ARCHIDOC, a bibliographic database of architectural heritage of the 19th and the 20th century. It contains about 67.000 bibliographic entries.

Besides these databases, which are highly heterogeneous in structure, the ministry has undertaken an important work concerning the development of *thesauri* for the consistent indexing of documents. The results of these efforts are, among others, the *Thesaurus de l'Architecture* [65] and the *Thesaurus de Dénomination*. These thesauri were constructed following the standard ISO 2788 [114] for the development of monolingual thesauri presented in Section 2.1.2, and in close collaboration with the developers of the *Art & Architecture* and the *RCHME* (Royal Commission of the Historical Monuments of England) thesauri for the establishment of *inter-thesaurus links*.

The descriptions stored in the databases are indexed using these thesauri, but, the main problem is that there is not a *unique* access (e.g. querying) interface for the databases mentioned previously. So, if a user requests data from more than one of the above databases, then she has to access each one of them, and process manually the obtained results. The need of a single entry point to the underlying data was a basic requirement which led to the development of the CI SGML DTD whose purpose is to provide a consistent way for describing cultural objects in SGML documents. It consists of folders that are classified in Content and Filling folders. Content folders are used to describe architectural artifacts (i.e., monuments, historical sites, etc.), mobile objects (i.e, paintings, sculptures, furniture, etc.) and thematic topics (i.e, family of buildings, series of objects, etc.). Filling folders are used to classify the content folders and they are organized on a geographical area (e.g., communal-departmental-regional). The approach used in the CI DTD is very structural: filling folders can contain each other based on their topological relationships and content folders can be constructed recursively.

The problem with the CI DTD is that it (i) contains a large number of elements, (ii) is not well structured and (iii) is not modular. Hence, it is very difficult to use it for creating descriptions of cultural objects. The objective of the project undertaken between the CNAM and the French Ministry of Culture was to create a *simple metadata schema* which describes the basic notions in the cultural domain and could be used by the different services of the ministry to *create metadata descriptions* in order to *index* their cultural artifacts. The idea was to develop a first prototype whose purpose was at a first time to *consistently describe* and *index* cultural objects.

In order to create the metadata schema, the basic concepts and their relationships to describe the cultural objects were first identified. In a second step, the approach described in Section 3.2 to integrate this schema with the available thesauri was followed. The result was a rich in semantics and structure schema that was subsequently used to create metadata descriptions of cultural objects.

3.5.1 ELIOT Metadata Schema

The ELIOT metadata schema, illustrated in Figure 3.17, was defined from scratch in collaboration with the persons in charge of the services of *Inventaire* and *Archéologie* of the French Ministry of Culture.

The ontology is rather simple and is comprised of nine concepts associated with twelve roles. It represents *cultural objects* (concept Objet d'Étude), which are documented by (role *documenté par*) documents (concept Document). Each object is situated (role *est située*) in a location (concept Localisation), and is found in some relative position (concept Position Relatif) with respect to an object of destination and an object of origin (roles *objet de destination* and *objet d'origine* respectively). An object consists of (role *est constitué de*) other objects, and is described by (role *décrit par*) a description (concept Description). This ontology describes also events (concept Événement) which in their turn are also associated with descriptions (role *associé à*). An event occurs (role *se passe en*) during a period (concept Période). A document refers other documents (role *réfère*) and is found in a place (concept Lieu). Finally, a location is associated with (role *georéfèrencement*) coor-

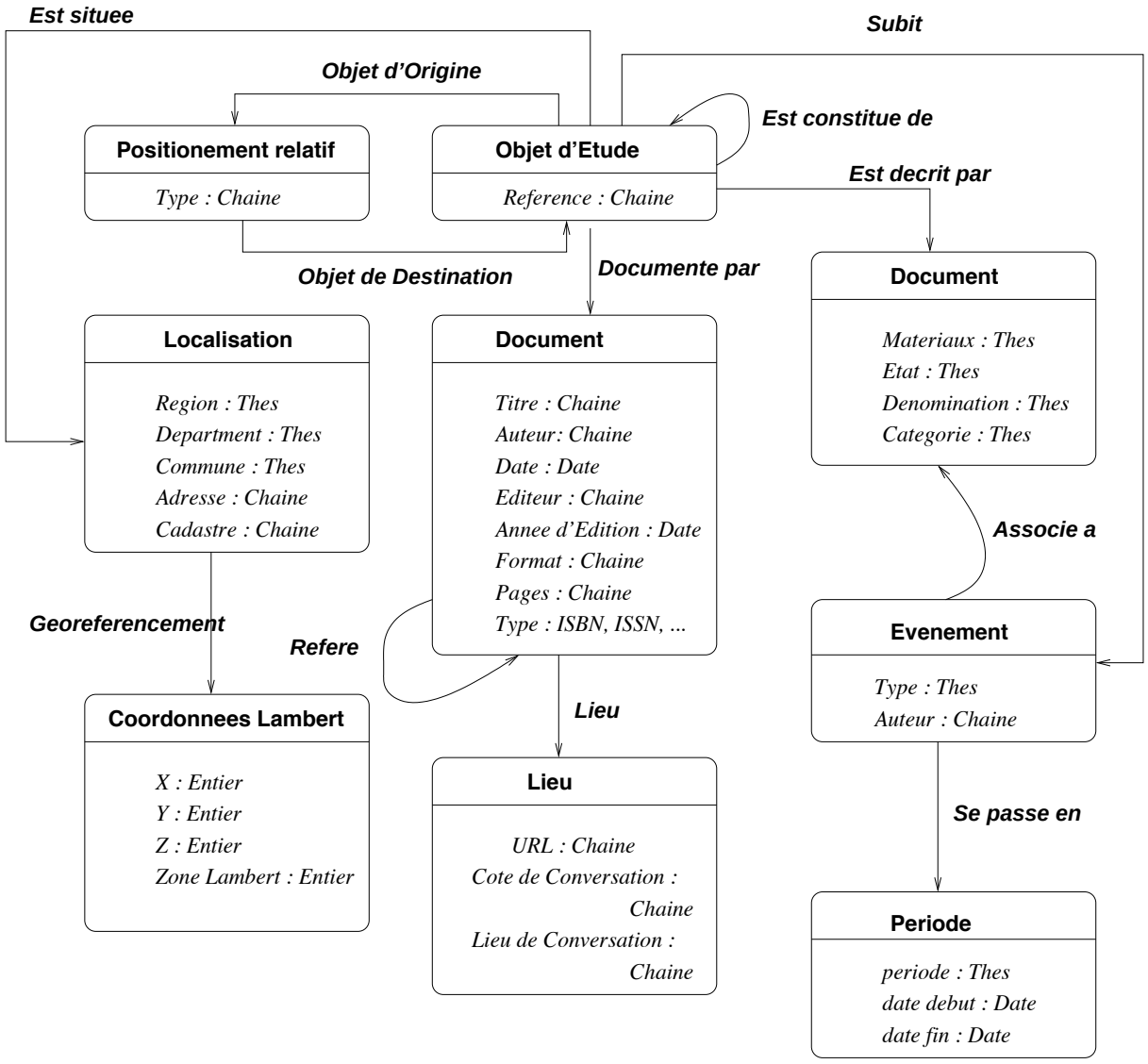

Figure 3.17: An Ontology for Cultural Artifacts

dinates (concept Coordonnés Lambert). The thesauri integrated with this schema are the *Thesaurus de Dénomination*, which contains terms for immobile objects, the *Thesaurus de la Rochelle*, which contains materials, and finally, the *Thesaurus de l'Architecture*.

3.5.2 System Architecture

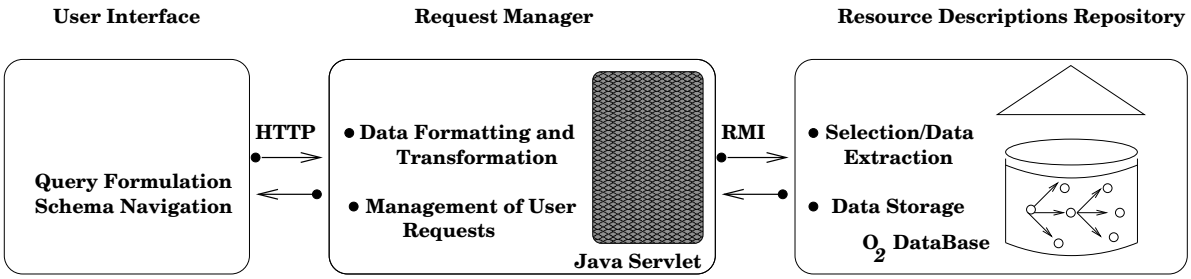

Figure 3.18: Eliot Architecture

The architecture of the ELIOT prototype [176] is shown in Figure 3.18. The prototype is built using a three level architecture. The basic modules of the system are the **User Interface**, **Request Manager** and the **Resource Descriptions Repository**.

The user is able to navigate in the ontology and formulate her queries using the **Query Interface**. Queries can be formulated using predefined HTML forms. The requests are then sent to the **Request Manager** which transmits them to the **Resource Descriptions Repository**. The latter is responsible for the storage of the schema and the source descriptions, and for answering the user requests. After evaluating the user requests, the answers are returned to the **Request Manager** who adds appropriate headers concerning the XSLT stylesheet that will be used for the transformation of the XML documents to HTML documents that will be presented to the navigator of the user.

The **Request Manager** is a *Java servlet* which runs on a Cocoon-enabled Apache Server. The communication between the **Request Manager** and the **Resource Descriptions Repository** is done using Java Remote Method Invocation (Java RMI) Objects. The OODBMS O_2 was used as a persistent data store. The programming language is Java, and the O_2 Java bindings were used to store Java objects in O_2 databases. O_2 Java bindings provide portability of Java programs without redefinitions of Java classes. The O_2 classes are created dynamically by importing Java classes. The ontology, the concept thesauri and source descriptions were represented using XML, and were loaded in the **Resource Descriptions Repository** using Xalan.

The choice of this three layer architecture offers a certain flexibility since we are not bound by the functionalities and implementations of the information providers.

In Figure 3.19 the query interface of ELIOT is presented. The HTML forms are created w.r.t. the metadata schema. Figure 3.20 shows a screendump of the results obtained for the user query that looks for the cultural objects (instances of the class Objet d' étude) which have been classified using the term *architecture religieuse*.

3.5.3 Bibliographic Notes on Labeling Schemes

Different labeling schemes have recently been proposed for various applications ranging from network routing, object programming, knowledge representation systems and latest XML search engines.

The existing labeling schemes can be classified in three categories. First *bit-vector* [217] schemes in which the label of a node is represented by a bit vector where a bit "1" at some position uniquely identifies the node in a DAG and each node inherits the bit positions of its immediate ancestors in top-down encoding (or descendants in bottom-up encoding). More compact variations of bit-vector based algorithms have been proposed in [11, 41, 128, 41].

Prefix-based encoding schemes, the Dewey scheme being the simplest example, which directly encode the parent of a node in a tree, as a prefix of its label. Applications of this algorithm to XML tree data have been proposed in [124, 61, 197]. Several variations have also been studied (see [108] for a comparative analysis and [96] for a recent survey), in order to provide more compact labels.

Finally, in *interval-based* encoding schemes [72, 73, 7, 141] the ancestor relationship of a node in a tree is encoded in this scheme as an inclusion of intervals computed using pre and post-order numbering of the tree.

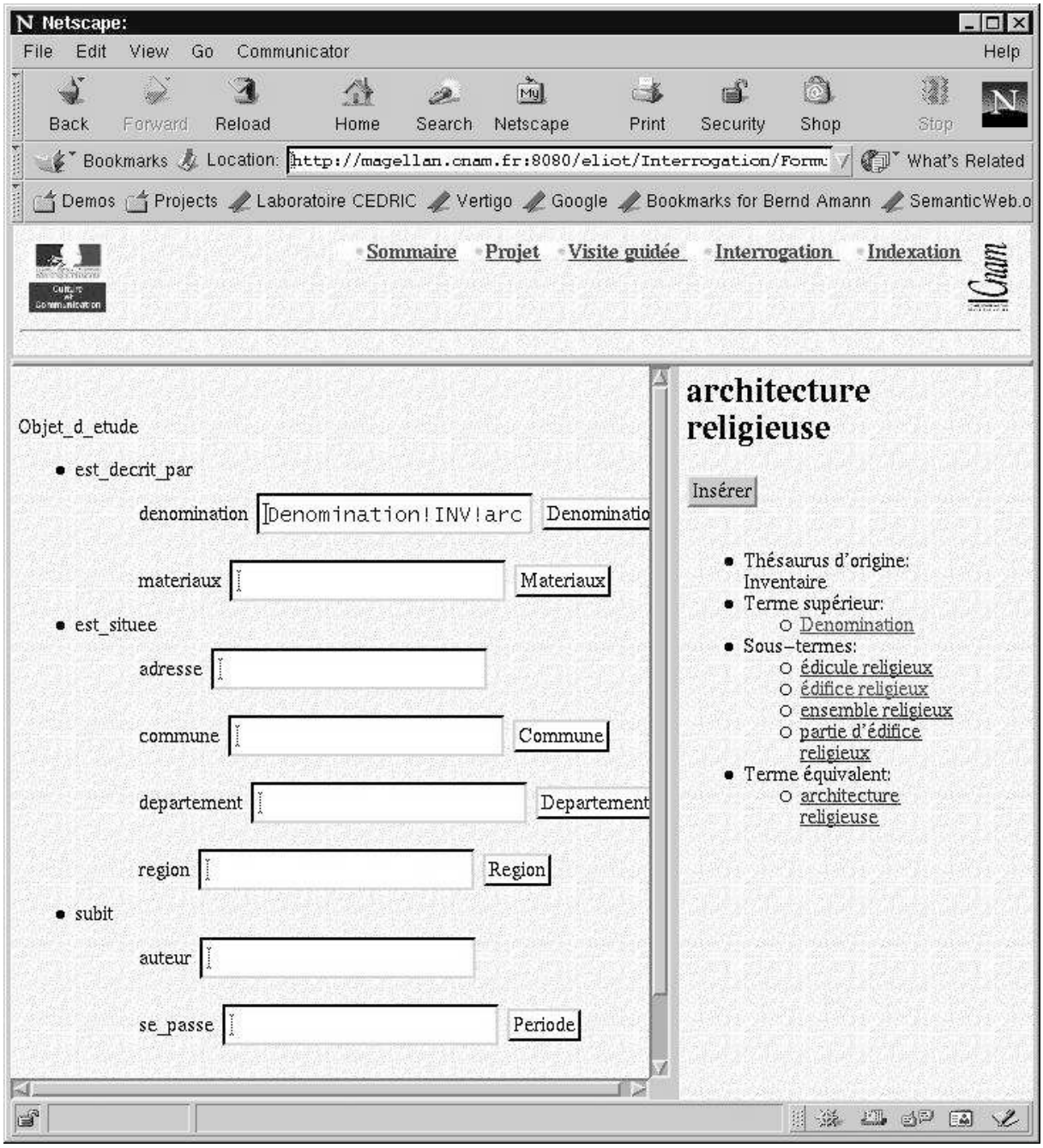

Figure 3.19: Query Interface

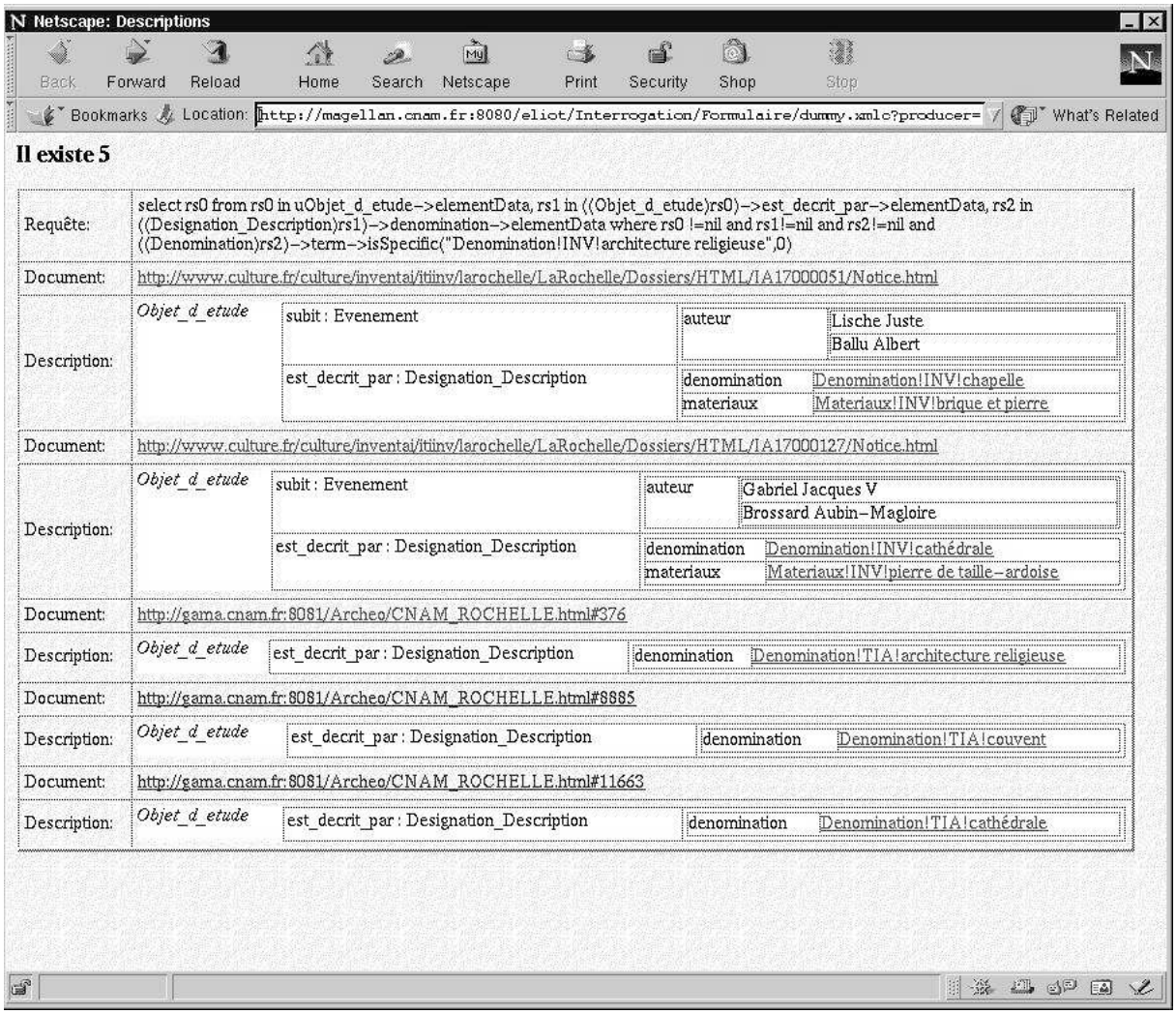

Figure 3.20: Result

Chapter 4

Integrating XML resources in $STYX$

In this chapter we present the $STYX$ approach for the *integration* and *querying* of *heterogeneous* and *autonomous* XML Web resources, based on the *mediator-wrapper* architecture proposed by Wiederhold in [216].

In $STYX$, sources to be integrated are associated with a *specific domain* or *application* of interest. The $STYX$ *mediator* offers a *unique* view of the underlying sources through a *global schema* which describes the basic notions in the domain. The schema is *not materialized* : the actual data resides in the sources. An XML source is published to the mediator by means of *mapping rules* specified by the source administrators. A user accesses the underlying sources by formulating queries in terms of the global schema and to obtain the answers, the query is translated into queries expressed in terms of the local sources schemas. This translation (or *rewriting*) is performed by the mediator using the established mapping rules. After evaluation, the results are returned to the mediator, combined and then presented to the user.

As illustrated in Section 2.3, there are two basic approaches for establishing mappings between the sources and the global schema. In the first approach, known as *global as view* (GAV), the global schema is defined as a *view* on the local sources schemas. In the second, known as *local as view* (LAV), a source is defined as a view of a *predefined* global schema.

We have chosen the *local as view (LAV)* approach as in [136, 149, 81] for mapping sources to the global schema. But in contrast to most of the systems that follow the LAV approach for integrating XML sources, the global schema is an *ontology* which contains *concepts*, *binary symmetric roles* and *inheritance (isa)* links between concepts. Sources are described by XML DTDs and are published to the mediator by means of *mapping rules*. These rules establish correspondences between XPath *location paths* defined on the XML structure and *ontology paths*. User queries are *tree* queries and are expressed in terms of the ontology.

To obtain answers for a user query, this must be evaluated against *all* sources. To obtain the results from a single source, we first *rewrite* the query into an XML query that is expressed in terms of the source's schema. There might be cases where results from different sources must be *combined* to obtain answers to the query. For this, it is necessary to *identify* that the objects obtained from different sources correspond to the same real world entity. For this purpose, we introduce the notions of *local* and *global keys*. Local keys are *physical keys*, used to identify objects obtained from a single document. On the other hand, global keys are *semantic* keys used to identify objects that originate from the same or from different sources.

Section 4.1 illustrates our approach by a motivating example taken from the cultural domain. Section 4.2 presents the data model of the $STYX$ mediator and the notion of *keys*. Section 4.3 introduces the *mapping language*, a simple but powerful language for describing XML resources

by associating *XPath location paths* [52] with *ontology paths*. In Section 4.4 we discuss how the $STyX$ database is obtained from the XML sources via the mapping rules. In Sections 4.5 and 4.6 we describe the $STyX$ *query language* and the *query rewriting*, *decomposition* and *execution plan generator* algorithms. Section 4.7 gives a discussion justifying our choices for the $STyX$ global schema and mapping language. Finally, we end the chapter with the description of the $STyX$ prototype validating our approach (Section 4.8).

4.1 System overview through a cultural example

4.1.1 XML resources

In this section we present the XML resources which we use throughout the presentation of our approach. Our assumption is that XML resources are described by XML Document Type Descriptions (DTDs). We use *cultural* XML resources to demonstrate our approach.

XML resource *http://www.paintings.com* is an XML resource about *painters* and their *paintings*. The XML DTD for the source is illustrated in Figure 4.1 and a document conforming to it is illustrated in Figure 4.2.

```
1.  <!ELEMENT Collection (Painter*)>
2.  <!ELEMENT Painter (Painting+, Sculpture+)>
3.  <!ATTLIST Painter Name CDATA #REQUIRED
4.                    Year CDATA #REQUIRED
5.  <!ELEMENT Painting (Image*,Technique?,museumID)>
6.  <!ATTLIST Painting Title CDATA #REQUIRED>
7.  <!ELEMENT Image EMPTY>
8.  <!ATTLIST Image URL #CDATA>
9.  <!ELEMENT Sculpture (Image*,Technique?,museumID)>
10. <!ATTLIST Sculpture Title CDATA #REQUIRED>
11. <!ELEMENT Technique #PCDATA>
12. <!ELEMENT museumID #PCDATA>
```

Figure 4.1: DTD for documents of *http://www.paintings.com*

Each *painter* (element `Painter`, line 2) is associated with one or more (occurrence indicator +) *paintings* (element `Painting`) and *sculptures* (element `Sculpture`). A painter has a *name* (attribute `Name`, line 3) and a *year of birth* (attribute `Year`, line 4). In turn, a painting and a sculpture (lines 5, 9) have a *title* (attribute `Title`, lines 6, 10), zero or more *images* (element `Image`, occurrence indicator *) an optional technique (element `Technique`, occurrence indicator ?) and an identifier (element `museumID`). An element of type `Image` is empty (line 7) and has a *URL* (attribute `URL`, line 8), that describes the location of the image. One can observe from the DTD that element types `Sculpture` and `Painting`, share the same structure.

XML resource *http://www.art.com* describes *museums* and the *objects* exposed in these museums. Its DTD is illustrated in Figure 4.3. Each museum (element `Museum`, line 2) has a *name* (element `MuseumName`) and a *city* (element `City`). Both elements are textual (lines 7,8). A museum is associated with one or more *artifacts* (element `Artifact`, occurrence indicator +, line 2). Each

```
<Collection>
  <Painter Name='Van Gogh' Year= '1800'>
    <Painting Title='Vincent's room at Arles'>
      <Technique>Oil On Canvas</Technique>
      <museumID>P12</museumID>
    </Painting>
    <Painting Title='Starry Night'>
      <museumID>P13</museumID>
    </Painting>
  </Painter>
  <Painter Name='Georges Braques' Year='1900'>
    <Painting Title='Black Fish'>
      <Technique>Oil On Canvas</Technique>
      <museumID>P17</museumID>
    </Painting>
  </Painter>
</Collection>
```

Figure 4.2: An XML document for source *http://www.paintings.com*

artifact (line 3) has a *title* (element `Title`), zero or more *images* (element `Image`, occurrence indicator `*`) and a year of creation (element `Year`). Element type `Image` is empty (line 4). It is associated with a *type* (attribute `type`, line 5) (for example `jpeg`, `gif` etc.) and a *location* (attribute `location`, line 6). Note that this source does not provide information about the artist who created the object.

```
1.   <!ELEMENT Museums (Museum)+>
2.   <!ELEMENT Museum (MuseumName, City, Artifact+)>
3.   <!ELEMENT Artifact (Title, Image*, Year)>
4.   <!ELEMENT Image EMPTY>
5.   <!ATTLIST Image type #CDATA #IMPLIED
6.                   location #CDATA #IMPLIED>
7.   <!ELEMENT MuseumName #PCDATA>
8.   <!ELEMENT City #PCDATA>
9.   <!ELEMENT Title #PCDATA>
10.  <!ELEMENT Year #PCDATA>
```

Figure 4.3: DTD for documents of *http://www.art.com* (art.dtd)

XML resource *http://www.all-about-art.com* is more complete than the previous ones. Its DTD is shown in Figure 4.4. This resource describes *paintings*, *painters* and *museums*. A *painting* (element `Painting`, line 2) has a *title* (element `Title`), and is associated with a *painter* by the XML attribute `painter` of type `IDREF` (line 3) and a *museum* by the XML attribute `museum` of type `IDREF` (line 5). Its place of creation is recorded by the attribute `place_of_creation` (line 4). In this source a *painter* (element `Painter`, line 7) is an empty element, and all information about a painter

is stored by means of XML attributes. A painter has a *name* (attribute `name`, line 8), a *date* and *place of birth* (XML attributes `date_of_birth` and `place_of_birth`, lines 9 and 10 respectively). A painter is *identified* by the attribute `painter_id`, of type `ID`. According to the XML standard, this identifier is *local* within the XML document : each painter element within a single document is associated with such a unique identifier. A museum is represented by the XML element (`Museum`, line 12) and has a *name* (element `MuseumName`) and a *city* (element `City`). Similar to painters, a museum has a unique identifier, represented using the attribute `museum_id` of type `ID` (line 13).

One can observe that the XML elements `Painting`, `Painter` and `Museum` are recorded as *top-level elements* in the DTD, and relationships between them are represented using *horizontal* (the `ID/IDREF` XML attribute mechanisms) instead of *hierarchical* relationships as done in the previous DTDs.

```
1.  <!ELEMENT Art (Painting|Painter|Museum)*>
2.  <!ELEMENT Painting (Title)>
3.  <!ATTLIST Painting painter #IDREF #REQUIRED
4.                     place_of_creation #CDATA #REQUIRED
5.                     museum #IDREF #IMPLIED>
6.  <!ELEMENT Title #PCDATA>
7.  <!ELEMENT Painter EMPTY>
8.  <!ATTLIST Painter name #CDATA #REQUIRED
9.                    date_of_birth #CDATA #REQUIRED
10.                   place_of_birth #CDATA #REQUIRED
11.                   painter_id #ID #REQUIRED>
12. <!ELEMENT Museum (MuseumName,City)>
13. <!ATTLIST Museum museum_id #ID #REQUIRED>
14. <!ELEMENT MuseumName #PCDATA>
15. <!ELEMENT City #PCDATA>
```

Figure 4.4: DTD for documents of *http://www.art.com* (art.dtd)

4.1.2 ST_YX Global Schema

The ST_YX global schema is an *ontology*, defined *independently* of the local sources schemas by domain specialists after some common agreement and defines the basic notions in the domain. In ST_YX we consider *ontologies* where the domain of interest is modeled by *concepts*, and *binary relations* between them. More specifically, *entities* in the domain are classified into *concepts*, *semantic relationships* between them are represented as *binary roles* between concepts, and their *properties* as *concept attributes*. In order to represent similarity of structures and to specify subset relationships between concepts, the *inheritance* relationship (*isa*) between *concepts* is used.

An example of a ST_YX ontology Let us give an example of a ST_YX ontology that we use throughout this manuscript to demonstrate our approach. The ontology shown in Figure 4.5 as a *directed labeled graph*, is an extract of the ICOM/CIDOC [75] Reference Model used for the documentation of cultural information. Nodes of the graph correspond to *concepts* and *atomic types* of the ontology. Concepts are connected by *binary roles*. *Attributes* connect concepts to atomic

types such as String, Integer, Float[1] etc. Both roles and attributes are shown as solid arcs in the schema. Concepts are also connected by *inheritance* (*isa*) links, depicted by dashed arcs in the schema.

Concepts : The ontology describes *concepts* such as Actor, Person, and Man_Made_Object. Concept Actor represents all actors, like organizations (groups of people, committees) or individuals (i.e. persons) who perform activities. Concept Person represents persons and is a *subconcept* of Actor. This is represented in Figure 4.5 by the *isa* dashed arc defined between the two concepts. Concept Event describes all different events, and its sub-concept Activity is there to represent the specific class of those events that produce some object. Concept Man_Made_Object collects all objects (and not only the ones made by some actor).

Roles : *Roles* are typed binary relationships between concepts. The fact that an actor (instance of concept Actor) performs an activity (instance of concept Activity) is represented by the role *carried_out*. This role is defined in concept Actor, which is called the *source* of the role. Concept Activity is called its *target*. The result of an activity is the creation of a man made object (instance of concept Man_Made_Object). This is represented by the role *produced*. An event happens at some place (instance of concept Place), defined by the role *took_place_at*. The occurrence of an event in some date (instance of concept Date) is represented by the role *took_place_in*.

Attributes : A concept can be associated with *attributes*. The *source* of an attribute is a concept, while its *target* is an *atomic type*. For example, a person (instance of concept Person) has a name (represented by the attribute *has_name*) which takes its values in the atomic type String. Similarly, an object (instance of concept Man_Made_Object) has a title represented by the attribute *has_title* which also takes its values in String.

Inverse Roles : One can observe that some of the roles in Figure 4.5 are depicted in parentheses. In fact, for each role its *inverse* is also defined. Our ontology is a *non-directed graph* but since it is used as a user interface for (i) the formulation of queries and (ii) the construction of the mapping rules, we give *directions* to roles to facilitate these tasks. Each role is replaced by two others : we choose arbitrarily one as the *default* role and we call the other its *inverse*. For example, for role *carried_out*, its inverse role is *carried_out_by*. The *source* of this role is concept Activity and its *target* is concept Actor.

Remarks : In our ontology, roles and attributes are *multi-valued* and *optional*. The *isa* relationship carries subset semantics and supports role and attribute inheritance. For example, the role *carried_out* defined in concept Actor is inherited to its subconcept Person.

4.1.3 Publishing XML sources in ST_YX

Now that we have introduced the XML resources to be queried and the ontology that describes the domain of discourse, let us see how these sources can be described in terms of the ST_YX ontology. An XML resource is published to the mediator by means of *mapping rules* that map *XPath location paths* [52] to *ontology paths*. The XPath location paths *address* fragments in the XML document structure. Ontology paths are *compositions* of ontology *concepts*, *roles* and *attributes*.

[1]The atomic types correspond to those defined in the XML Schema [26] standard.

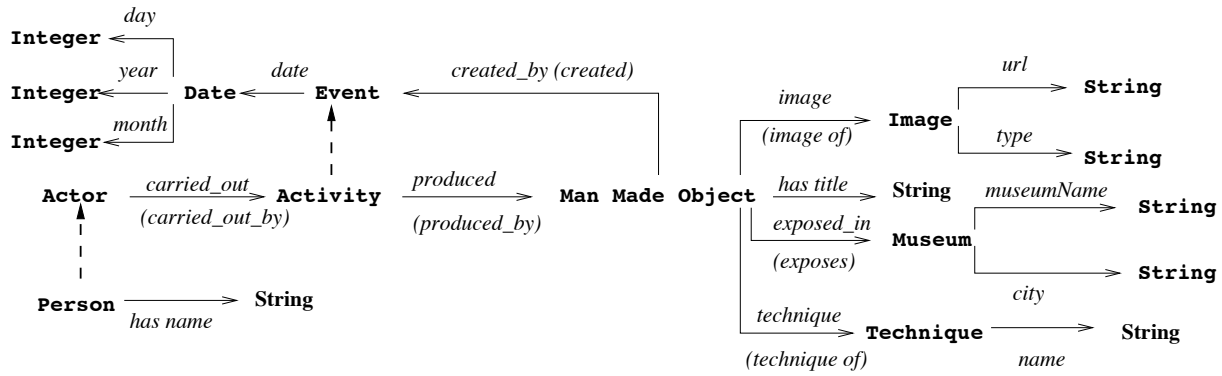

Figure 4.5: An Ontology for Cultural Artifacts

Let us illustrate the *STYX* mapping language by considering the XML resources presented in Section 4.1.1 and the example ontology in Section 4.1.2.

Publishing XML source *http://www.paintings.com* This source, whose DTD is illustrated in Figure 4.1 provides information about *painters* and their *paintings*. If we want to describe the contents of this source in terms of our ontology, we would like to say that *painters* are persons, and *paintings* are *man made objects*. To express these two assertions, the following two rules are written :

A_1:	`http://www.paintings.com/Collection/Painter` as u_1	→	Person
A_3:	`http://www.paintings.com/Collection/Painter/Painting` as u_3	→	Man_Made_Object

Each rule has a left hand side (LHS) and a right hand side (RHS). The right hand side is an *ontology path*. The left hand side is composed of (i) an *XPath location path* evaluated on some URL (or on some variable) and (ii) a *variable declaration*. The XPath location path is called the *source path* of the rule.

The first rule states that instances of concept Person are obtained by evaluating XPath `Collection/Painter` on the documents in URL *http://www.paintings.com*. The XPath expression returns `Painter` elements which are *children* of `Collection` elements. Variable u_1 *binds* these elements and is called the *bound* variable of the rule. The statement *'as* u_1*'* is a *variable declaration*.

The second rule states that instances of concept Man_Made_Object are obtained from the source by evaluating XPath `Collection/Painter/Painting` on documents in URL *http://www.paintings.com*. The obtained `Painting` elements (bound in variable u_3) are children of `Painter` elements, themselves children of `Collection` elements. Rules A_1 and A_3 are called *absolute rules* : their source path is evaluated on a URL.

Suppose now that we need to state that the names of persons (values of attribute *has_name*) are obtained from the attribute `Name` of `Painter` elements :

A_2:	u_1`/@Name` as u_2	→	has_name

Recall that variable u_1 binds `Painter` elements, instances of concept Person. The XPath `@Name`, when evaluated on instances of variable u_1, returns XML attribute nodes (XPath axis '@') which are bound by the rule to variable u_2. In this rule, variable u_1 is called the *root* variable of the rule. Rule A_2 is a *relative rule* : its source path is evaluated on a variable.

The reader may observe that we map XML nodes to string values of the ontology. This is done by *casting* these XML nodes to their *values* using the special-purpose functions of the XPath language [52] like `string()` which returns the string value of an XML node.
In the previous rule, a pair (x, y) where x is an instance of variable u_1 and y is an instance of variable u_2 becomes an instance of attribute *has_name*.

Note that to follow from instances of variable u_1 (obtained from rule A_1), the ontology path of rule A_2, the *composition* of the ontology paths of rules A_1 and A_2 must also be a *path* in the ontology. This is true in our case since attribute *has_name* is defined in concept Person. By concatenating the source paths and the ontology paths of those rules, we obtain a new rule $A_1.A_2$:

$A_1.A_2$: `http://www.paintings.com/Collection/Painter/@Name` as u_2 → Person.has_name

Consider again rule A_3. This rule returns instances of concept Man_Made_Object, obtained by evaluating the XPath `Collection/Painter/Painting` on documents in URL *http://www.paintings-.com*. Suppose that we know from the source that the paintings organized under their respective painter are those that have been *created* by the painter and not those that *influenced* the painter. In order to make this information explicit, we write the rule illustrated below :

A_3: u_1/`Painting` as u_3 → carried_out.produced

The source path of the rule is `Painting`, evaluated on the instances of variable u_1 (which are obtained by rule A_1). The `Painting` elements obtained by the rule are bound in variable u_3.

Rule A_3 returns XML elements of type `Painting` which are instances of concept Man_Made_-Object (the *target* of the role *produced* is concept Man_Made_Object). These instances are obtained from instances of concept Person reached by following the roles in the *path carried_out.produced*. Here the ontology path of the rule is a *role path* : a path defined by the composition of roles *carried_out* and *produced*.

This rule not only returns instances of concept Man_Made_Object, but defines instances of the *role path carried_out.produced* which is a *derived role* (i.e. it does not exist in the initial ontology). These instances are pairs (x, y) where x is an instance of variable u_1 (i.e. instance of concept Person) and y an instance of variable u_3 (i.e. instance of concept Man_Made_Object).

Notice here that if rules A_1, A_3 are concatenated, we obtain rule $A_1.A_3$:

$A_1.A_3$: `http://www.paintings.com/Collection/Painter/Painting` as u_3 → Person.carried_out.produced

Observe the ontology path of the rule : it is composed of a *concept* (Person) and a *role path* (*carried_out.produced*). This ontology path is called *concept path*, and returns instances of concept Man_Made_Object that have been created by *some* instance of concept Person. It defines a *derived concept* which is a subconcept of Man_Made_Object. Observe that the XPath `Collection/-Painter/Painting` is evaluated on URL *http://www.paintings.com* and not on a variable which binds instances of concept Person. For this concept path, we know that there exists some instance of concept Person who created the man made object, but this instance is not considered by the rule.

The set of mapping rules by which source *http://www.paintings.com* is published are shown in Figure 4.6.

A_1:	`http://www.paintings.com/Collection/Painter` as u_1	→	Person
A_2:	u_1/`@Name` as u_2	→	has_name
A_3:	u_1/`Painting` as u_3	→	carried_out.produced
A_4:	u_3/`@Title` as u_4	→	has_title
A_5:	u_3/`Image` as u_5	→	image
A_6:	u_5/`@URL`	→	url
A_7:	u_1/`@Year`	→	born.took_place_in.year
A_8:	u_1/`Sculpture` as u_3	→	carried_out.produced
A_9:	u_3/`museumID` as u_6	→	museumIdentifier

Figure 4.6: Set of Mapping Rules for source *http://www.paintings.com*

Publishing XML source *http://www.all-about-art.com* Elements in this source are organized as top level elements, and are referenced using XML attributes of type `ID/IDREF`. The mapping rules for this source are shown in Figure 4.7.

Consider mapping rule C_1. This rule states that instances of concept Man_Made_Object are obtained when evaluating XPath `Art/Painting` on documents of URL *http://www.all-about-art.com*. XML fragments obtained by this rule are bound in variable w_1.
Observe now rule C_5. This rule states that instances of concept Museum (the target of role *exposed_in*), are obtained by evaluating XPath `id(@museum)` on instances of variable w_1. In the source path of the rule, the function `id()` from the core library of XPath is used [52]. This function takes as a parameter a string s, and returns the XML element in the document which has an attribute of type `ID` with value s. In rule C_5, the parameter passed is the value of attribute `museum` of the `Painting` elements.
Rule C_6 states that the painters who have created the paintings are obtained by evaluating XPath `id(@painter)` on instances of variable w_1. Recall that the ontology path of the rule is *produced_by.carried_out_by* which uses the inverse roles of the ontology.

C_1:	`http://www.all-about-art.com/Art/Painting` as w_1	→	Man_Made_Object
C_2:	w_1/`Title`	→	has_title
C_3:	w_1/`@date_of_creation`	→	created.took_place_in.year
C_4:	w_1/`@place_of_creation`	→	created.took_place_at.placeName
C_5:	w_1/`id(@museum)` as w_2	→	exposed_in
C_6:	w_1/`id(@painter)` as w_3	→	produced_by.carried_out_by
C_7:	`http://www.all-about-art.com/Art/Painter` as w_3	→	Person
C_8:	w_3/`@name`	→	has_name
C_9:	w_3/`@date_of_birth`	→	born.took_place_in.year
C_{10}:	w_3/`@place_of_birth`	→	born.took_place_at.placeName
C_{11}:	`http://www.all-about-art.com/Art/Museum` as w_2	→	Museum
C_{12}:	w_2/`MuseumName`	→	museumName
C_{13}:	w_2/`City`	→	city

Figure 4.7: Set of Mapping Rules for source *http://www.all-about-art.com*

Summary An XML source is published in the $STyX$ mediator by *mapping rules*.
A mapping rule consists of an *XPath location path* (called *source path*) which is evaluated on a variable or on a URL, and an *ontology path*. The variable on which the XPath location path is

evaluated is called the *root variable* of the rule. The XML nodes (elements or attributes) obtained are bound to a variable which is called the *bound variable* of the rule. The *ontology path* of the rule is defined in terms of the ST_YX ontology.

We distinguish between *absolute* and *relative* rules. A rule is called *absolute* if its source path is evaluated on a URL, and *relative* if it is evaluated on some variable.

A set of mapping rules by which a source is published in the ST_YX mediator is called a *mapping*. A source can be published by more than one mappings in ST_YX.

A rule returns instances of a *concept*, an *attribute*, a *role* or an *ontology path*. More specifically, an absolute mapping rule whose ontology path is a concept c or a concept path p, returns instances of c or of the concept reached by p.

A relative mapping rule whose ontology path is a *role* or an *attribute* path returns (i) instances (values) of the concept (atomic type) reached by the path and (ii) instances of the role/attribute path.

Two rules R_1 and R_2 can be *concatenated* when the root variable of R_2 is the bound variable of R_1 and the *composition* of their ontology paths is a path in the ontology.

Use of variables in mapping rules The use of *variables* in the mapping rules is a tool that serves different purposes :

- First, it allows us to reduce the number of mapping rules, sometimes quite drastically. In general, a set of k_1 rules that bind variable u, and a set of k_2 rules that use it can be extended by a set of $k_1 * k_2$ rules by applying *concatenation*. That is, the resulting set of mapping rules after expansion may be exponentially larger than the set of rules with variables.
- Second, the use of variables enables one (i) to map different XPaths to the same variable and (ii) to separate contexts in which different mapping rules can apply.

  1. Let us consider the case where different XPaths are mapped to the same variable. Consider for example XML source *http://www.paintings.com* illustrated in Figure 4.1, and the mapping rules by which it is published, shown in Figure 4.6. Notice that elements of type `Painting` and `Sculpture` have exactly the same structure : they have an attribute `Title`, one or more sub-elements of type `Image`, an optional `Technique` subelement and a `museumID` subelement. Notice that rules A_3 and A_8, which return elements of type `Painting` and `Sculpture` respectively, are both assigned the same variable u_3. By this multiple assignment, rules A_4, A_5 and A_9 can be applied to all fragments obtained by rules A_3 and A_8.
  2. Consider the case where we *separate* contexts in which different mapping rules can be applied. There are two issues here :
     (a) to distinguish between XML fragments which are mapped to the *same ontology path* but have *different structures*, and
     (b) to distinguish between the *different ontology paths* to which the *same XML fragment* is mapped to.

     For the first case, consider the example of a source which provides information about painters and sculptors. Elements `Painter` and `Sculptor` have different structures : the former is associated with the person's year of birth and the latter with her place of birth. The mapping rules are illustrated below.

R_1:	URL/Painter as x_1	→	Person
R_2:	URL/Sculptor as x_2	→	Person
R_3:	x_1/@year	→	born.took_place_in.year
R_4:	x_2/@place	→	born.took_place_at.placeName

Notice that Painter and Sculptor elements are mapped to the same concept (Person). Nevertheless, different variables (x_1, x_2) are used in the mapping rules to bound these XML elements to depict their different structures.

For the second case, consider a source that describes museums, where each museum is located in a city, and has a name. A museum also produces some objects, and in this case is considered as an actor. The mapping rules for this source are illustrated below.

S_1:	URL/Museum as x_1	→	Museum
S_2:	URL/Museum as x_2	→	Actor
S_3:	x_1/@name	→	museumName
S_4:	x_2/Artifact	→	carried_out.produced

The use of different variables in the mapping rules above separate the context in which a Museum element is considered as a *museum* and the context in which it is considered as an *actor*.

4.1.4 Query Evaluation

In this section we present query evaluation in $STYX$. The user exploits the set of sources by formulating her queries in terms of the *global schema concepts*, *roles* and *attributes*. More specifically, the user views the $STYX$ mediator as a single database of objects, where an object is represented by an XML fragment, without being aware of where these objects originate from.

User queries In $STYX$ user queries are *tree queries* expressed in an OQL-like syntax, and formulated in terms of the ontology. An example of a user query is query Q_1 illustrated in Table 4.1. This query asks for *"the titles of the objects created by 'Van Gogh"'*. Query variables are defined in the query **from** clause. Variable x_1 is the *root* variable of the query and ranges over instances of *concept* Person. Variable x_2 binds the values obtained from instances of variable x_1 when following attribute *has_name*. The values reached from instances of variable x_1 by following the *path carried_out.produced* are bound in variable x_3. Finally, variable x_4 ranges over the values obtained by following attribute *has_title* from instances of variable x_3.

Q_1:	**select**	x_4
	from	Person x_1, x_1.has_name x_2,
		x_1.carried_out.produced x_3,
		x_3.has_title x_4
	where	x_2 = "Van Gogh"

Table 4.1: Query Q_1

The *ontology paths* in the query **from** clause are called *binding paths* of the variables. Only the root variable of the query ranges over instances of a *concept* or a *concept path*. All other variables range over atomic values or objects (i.e. *concept instances*) obtained by following *role* or *attribute*

paths. Variables appearing in the **from** clause are *existentially quantified.* The query **where** clause is a conjunction of simple comparison predicates.

The answer to a query is a *set of tuples* of the form $(x_1 : v_1, x_2 : v_2, \ldots x_n : v_n)$ where x_i is a variable that appears in the query **select** clause, and v_i is an atomic value or an object.

Evaluating Queries Query evaluation in *STyX* is done as follows : given a user query q and a set of sources published using mapping rules, we must get all answers that satisfy q. A source s returns only a *subset* of the answers for q. To get additional answers, q must be evaluated over *all* sources.

Let us consider query evaluation for a specific source s. The query is expressed in terms of the ontology. To evaluate it on the source, the query has to be *rewritten* into an XML query expressed in terms of the source's schema, and in the query language supported by the source.
There are two cases when performing this rewriting (translation) :

- the translation is *full*, i.e. the source is able to provide answers for the *whole* query;
- the translation is *partial*, i.e. the source answers only a *part* of the query.

In the last case, to get answers for the whole query we must *decompose* the query into (i) the subquery that the source can answer, and (ii) the subqueries that the source cannot answer, which can be possibly answered by other sources.

Rewriting Queries The *rewriting algorithm*, presented in detail in Section 4.6.1, is a *two phases algorithm* :

- In the first phase, *mapping rules* that provide answers for the query variables are found. For this, the *query variables binding paths* are *matched* with the *ontology paths* of the *mapping rules.* The result of this phase is a *(set of) query variables to rules binding(s).* We distinguish between two kinds of bindings :
  1. A *full* binding is one that binds all query variables.
  2. A *partial* binding binds a proper subset of the query variables.
- In the second phase, given the *bindings* calculated in the first phase, the query is rewritten into one or more XML queries to be sent to the source for evaluation. For this rewriting, first the *binding path* of each variable is substituted by the *source path* of its corresponding rule in the binding. Second, a simple syntactical transformation of this query into the query language supported by the source is performed.

Example 4.1.1 *Consider query* Q_1 *illustrated in Table 4.1 and the mapping rules for source* http://-www.paintings.com *shown in Figure 4.6. First of all, we want to obtain instances for variable* x_1 *which is bound to concept* Person. *We look for an* absolute *rule, whose right hand side is concept* Person *or a subconcept thereof. Such a rule is* A_1*, hence variable* x_1 *is bound to* A_1*. Instances of variable* x_2 *(reached by instances of* x_1 *by following attribute* has_name*) are found by rule* A_2*. Notice that the ontology path of the rule is* has_name *and is equal to the binding path of variable* x_2*. Moreover, rule* A_2 *can be applied on instances of variable* x_1 *since its root variable is the same as the bound variable of* A_1 *(*A_1 *returns instances of variable* x_1*).*

Similarly, instances for variable x_3 *are found by rules* A_3 *and* A_8 *and instances for variable* x_4 *(reached from instances of* x_3 *by traversing attribute* has_title*) are found by rule* A_4*. The steps of the process are illustrated in Figure 4.8.*

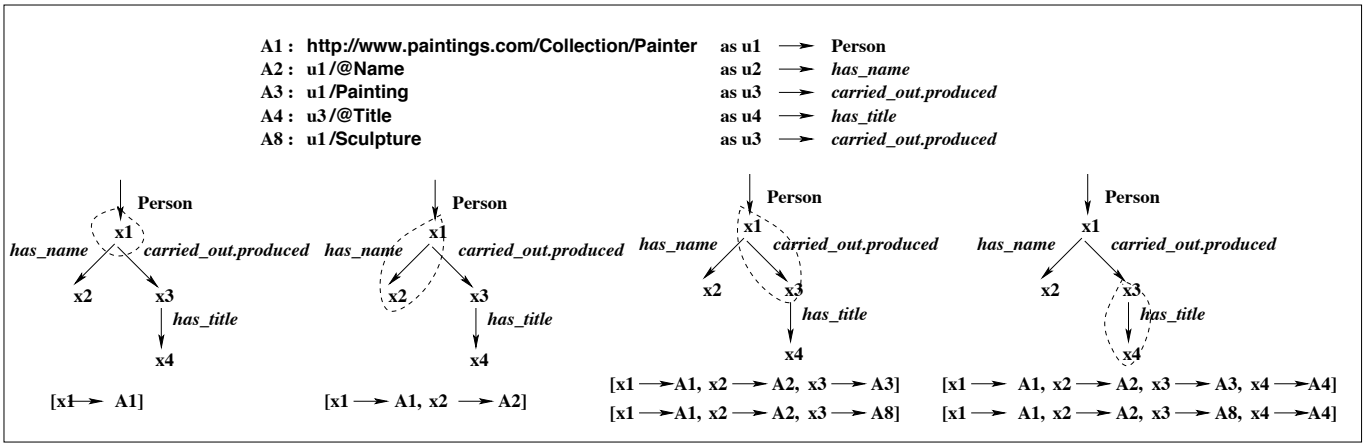

Figure 4.8: Calculating Bindings for query Q_1 and source http://wwww.paintings.com

Finally, the following two bindings are obtained :

$$[x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_3, x_4 \mapsto A_4] \tag{4.1}$$

$$[x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_8, x_4 \mapsto A_4] \tag{4.2}$$

To rewrite the previous query into an XML query, in a first step the query variables binding paths are substituted by the source paths of the rules with which these variables are associated in the binding. The resulting query for binding 4.1 is shown below.

$Q_1(a)$:	**select**	x_4
	from	`http://www.paintings.com/Collection/Painter` x_1,
		x_1`./@Name` x_2, x_1`./Painting` x_3,
		x_3`./@Title` x_4
	where	x_2 = *"Van Gogh"*

The query in this intermediate form can be rewritten in a more or less straightforward manner to an XOQL [9], XML-QL [71], XQuery [44] query or to an XPath [52] expression. For example, the XQuery expression for query $Q_1(a)$ is illustrated below.

```
FOR     $x1 IN
        document("http://www.paintings.com")/Collection/Painter,
        $x2 IN $x1/@Name, $x3 IN $x1/Painting,
        $x4 IN $x3/@Title,
WHERE   $x2 = "Van Gogh"
RETURN
        <RESULT>
          <Person>
            <carried_out.produced>
              <has_title>$x4</has_title>
            </carried_out.produced>
          </Person>
        <RESULT>
```

Remark that in the **RETURN** *clause of the XQuery expression the result is typed using the binding path of the variable in the* **from** *clause of the initial query. The result of this XQuery expression when evaluated on the XML document illustrated in Figure 4.2 is the XML document shown below.*

```
<RESULT>
  <Person>
    <carried_out.produced>
      <has_title>Starry Night</has_title>
    </carried_out.produced>
  </Person>
  <Person>
    <carried_out.produced>
      <has_title>Vincent's room at Arles</has_title>
    </carried_out.produced>
  </Person>
</RESULT>
```

From the obtained XML document, one can define two labeled directed trees *of* objects *and* values. *In such trees, objects are created for elements (XML fragments) and are labeled with* concepts. *Edges between objects are labeled with* roles, role paths, attributes *or* attribute paths. *Edges labeled with attributes or attribute paths connect objects in the tree to values. The trees defined for the XML document illustrated above are shown in Figure 4.9.*

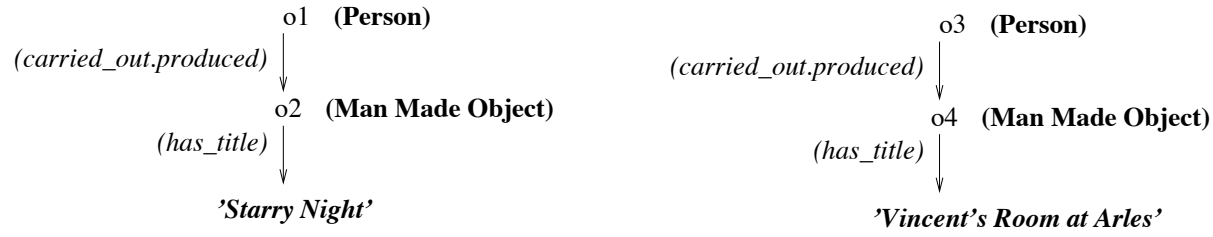

Figure 4.9: Two labeled trees of objects

In Figure 4.9 the concepts, with which objects are labeled, and role/attribute (paths), with which edges are labeled, are shown within parentheses. Object o_1 *created for the first element* `Person` *in the resulting XML document is labeled with concept* Person. *Object* o_2 *is created for element* `carried_out.produced` *and is labeled with concept* Man_Made_Object *(the target of the path* carried_out.produced *in the ontology). The edge between objects* o_1 *and* o_2 *is labeled with the role path* carried_out.produced. *Edge* has_title *is created for element* `has_title` *which connects object* o_2 *to the value 'Starry Night'. Finally, the result of the query is the set of tuples :* $\{(x_4 :' Starry\ Night'), (x_4 :' Vincent's\ room\ at\ Arles')\}$.

Example 4.1.2 *Query* Q_2 *is an extension of query* Q_1 *illustrated in Table 4.1 which asks for* "the title of the objects created by Van Gogh and the names and cities of the museums where the objects are exposed.".

Q_2:	**select**	x_4, x_6, x_7
	from	*Person* x_1, x_1.*has_name* x_2,
		x_1.*carried_out.produced* x_3,
		x_3.*has_title* x_4,
		x_3.*exposed_in* x_5,
		x_5.*museumName* x_6, x_5.*city* x_7
	where	x_2 = *"Van Gogh"*

Take again source http://www.paintings.com. *The source can only answer subquery* Q_1 *of* Q_2. *We see that no answers can be found for variable* x_5 *: there does not exist a rule whose ontology path is equal to* x_5*'s binding path. The same holds for variables* x_6 *and* x_7.

Query Decomposition Source *http://www.paintings.com* provides *partial answers* for query Q_2 : for an instance of variable x_3 (instance of concept Man_Made_Object) obtained by the source, we cannot obtain the museum (and consequently its name and city) where it is exposed. In this case, the query to be sent to the other sources for evaluation is the one looking for *"the name and city of the museum of man made objects"*.

Given a *partial binding*, the process of identifying (i) the subquery for which the source gives full answers (called *prefix* query), and (ii) the subquery which the source cannot answer (called *suffix* query)[2], is called *query decomposition*. The results obtained from the two queries must then be *joined* at the mediator site.

Example 4.1.3 *In Example 4.1.2, the prefix query to be evaluated by the source* http://www.-paintings.com *is* $Q_2(a)$ *illustrated in Table 4.2. The suffix query sent to the other sources for evaluation is* $Q_2(b)$ *illustrated in Table 4.3. Notice that in* $Q_2(b)$ *variable* x_3 *(the root variable of the query) ranges over instances of concept* Man_Made_Object *which is the target concept of the binding path of* x_3 *(*carried_out.produced*) in query* Q_2.

$Q_2(a)$:	**select**	x_4
	from	Person x_1,
		x_1.has_name x_2,
		x_1.carried_out.produced x_3,
		x_3.has_title x_4
	where	x_2 = 'Van Gogh'

Table 4.2: Query $Q_2(a)$

$Q_2(b)$:	**select**	x_6, x_7
	from	Man_Made_Object x_3, x_3.exposed_in x_5,
		x_5.museumName x_6,
		x_5.city x_7

Table 4.3: Query $Q_2(b)$

[2]There might exist more than one subqueries that the source cannot answer.

Identifying and Joining Objects : In the previous example, whereas $Q_2(a)$ returns titles of objects and $Q_2(b)$ returns museums and cities, it is possible to display for each instance of variable x_3 the title and the museum name and city, only if a *join* is performed between the result of both queries. To perform this join, objects must be *identified*.

Identifying Objects In our context, objects are identified by *keys*. Keys is a standard way to identify *information entities* in terms of their *properties* (e.g. attributes in a relational table). In our context we distinguish between two kinds of keys for objects : *local* and *global* keys.

The XML attribute of type `ID` and the *position* of an XML fragment in a document are *local keys*. These identify objects (represented by XML fragments) that originate from the *same* document. These local keys can only be considered as *internal* pointers within a single XML document and cannot be used to identify objects that originate from different documents.

In contrast to local keys, a *global key* is defined at the ontology level, *independently* of the XML sources. *Global* keys are *semantic keys* defined in *concepts*. A key for a concept is a set of *attribute paths* originating from it. For example, an instance of concept Man_Made_Object is identified by its *museum identifier* [3]. In this case the key for the concept consists of the attribute *museumIdentifier*. Finally, a concept can be associated with more than one global keys.

To be able to obtain the key values for an instance of variable x, we 'extend' the initial user query with attribute paths defined in keys. In Example 4.1.3, the key value of an instance of variable x_3 (instance of concept Man_Made_Object) is obtained by requesting the value of the attribute *museumIdentifier*. For example, the prefix query $Q_2(a)$ becomes $Q_2(c)$ when requesting the values for attribute *museumIdentifier* for variable x_3.

$Q_2(c)$:	**select**	**t**, x_4
	from	*Person* x_1,
		x_1.*has_name* x_2,
		x_1.*carried_out.produced* x_3,
		x_3.*has_title* x_4,
		x_3.**museumIdentifier t**
	where	x_2 = *'Van Gogh'*

The XQuery that is sent to the source for evaluation is illustrated below.

[3] Identifiers of artifacts provided by museums can be used as global keys. For example, cultural artifacts in the museum of Louvre are associated with such identifiers. Other sources can use this information to refer to these objects. The *Object-ID* [199] initiative of the Getty Institute aims at developing a standard methodology for creating identifiers of cultural objects.

$XQ_2(c)$	**FOR**	$\$x_1$ **IN**
		document("http://www.paintings.com")/Collection/Painter,
		$\$x_2$ **IN** $\$x_1$/@Name, $\$x_3$ **IN** $\$x_1$/Painting,
		$\$x_4$ **IN** $\$x_3$/@Title,
		$\$t$ **IN** $\$x_3$/museumID,
	WHERE	$\$x_2$ = *'Van Gogh'*
	RETURN	
		$< RESULT >$
		$< Man\_Made\_Object >$
		$< has\_title >$ $\$x_4$ $< /has\_title >$
		$< museumIdentifier >$ $\$t$ $< /museumIdentifier >$
		$< /Man\_Made\_Object >$
		$< /RESULT >$

The result of a query extended with keys is a set of XML documents. For example, the XML document obtained from the evaluation of XQuery $XQ_2(c)$ against the document in Figure 4.2 is illustrated in Figure 4.10.

```
<RESULT>
  <Man_Made_Object>
    <has_title>Starry Night</has_title>
    <museumIdentifier>P12</museumIdentifier>
  </Man_Made_Object>
  <Man_Made_Object>
    <has_title>Vincent's room at Arles</has_title>
    <museumIdentifier>P13</museumIdentifier>
  </Man_Made_Object>
 </RESULT>
```

Figure 4.10: Intermediate XML query results

Joining Objects Given that objects obtained from the evaluation of the prefix and suffix queries are associated with their keys, it is possible to *join* them. In the relational model the *join* requires that the relations whose tuples are joined, agree on some common attributes. If these attributes constitute the *key* for the relations then this join is an *identity-based* join, called also *object fusion*. The result of the join between two objects o_1 and o_2 is a *new object* whose structure is obtained by 'merging' the structures of o_1 and o_2. Join is more formally discussed in Section 4.2.5.

Equivalent Queries As mentioned earlier, the ontology is a graph, and therefore the user can formulate a query using any of the roles in the ontology. For example query Q_2 which requested *"the titles, the names and cities of the museum where objects created by Van Gogh are exposed"* can be also formulated as query Q_2' illustrated below. Recall that the root variable of query Q_2 is bound in concept Person whereas in query Q_2', the root variable is bound in concept Museum. The binding paths of certain variables in Q_2 are inversed in Q_2'.

Q'_2	**select**	x_4,x_6, x_7
	from	Museum x_5,
		x_5.exposes x_3,
		x_5.museumName x_6,
		x_5.city x_7,
		x_3.has_title x_4,
		x_3.produced_by.carried_out_by x_1,
		x_1.has_name x_2
	where	x_2 = "Van Gogh"

If we try to rewrite this query for source *http://www.all-about-art.com*, we see that instances of concept Museum (to which variable x_5 is bound) are obtained by rule C_{11}, but no instances are obtained for the remaining variables. Although the source provides the necessary information (describes persons that create man made objects and the museums where these objects are exposed) we cannot obtain it by applying the rewriting algorithm illustrated previously since the user query is expressed using the inverse roles of those used in the mapping rules.
To resolve this problem there exist two choices : either to inverse the roles in the query, or to inverse the direction of the mapping rules statically, before rewriting.

Let us look at the second solution. In this case we must inverse not only the ontology paths of the mapping rules, but also the XPath location paths. As far as location paths are concerned, there are cases where we can simply use the parent/ancestor axis (if child/descendant axis are used) or vice versa. For example, consider the two rules A_1 and A_3 of source s_1 illustrated below.

A_1:	`http://www.paintings.com/Collection/Painter` as u_1	$\rightarrow$	Person
A_3:	u_1/`Painting` as u_3	$\rightarrow$	carried_out.produced

To find from the instances of man made objects their creators, we must write the following mapping rule :

A_3^-:	u_3/`parent::Painter` as u_1	$\rightarrow$	produced_by.carried_out_by

Given the set of mapping rules, one calculates only the 'inverse' of the relative mapping rules. More precisely, one can inverse the mapping rules whose ontology paths are role paths. Absolute mapping rules (i.e. those whose source paths are evaluated on the URL of the rule) are not inversed. Authors in [167] demonstrate how an XPath location path using child (descendant) axes can be rewritten into an equivalent path using the parent (ancestor) axes and vice versa. In our context, the inverse rule of a mapping rule of the form $R : a/q \; as \; b \rightarrow p$ is rule $R^- : b/q' \; as \; a \rightarrow p^-$ where q' is the equivalent XPath location path of q defined using the inverse axis of the ones used in q and p^- is the inverse path of p.

In the first solution, out of the initial query, a set of *equivalent queries* is computed. This rewriting is performed using the *inverse* roles of the ones present in the user query. To achieve this, each query variable (except those bound in the initial query to an atomic type), are considered as a root of an equivalent query. Consequently, if the initial query contains n variables that are not bound to an atomic type, then n such tree queries are specified where each of those variables is considered as root. For example for query Q'_2 illustrated previously, the set of equivalent queries is

illustrated below.

Q_2''	**select**	x_4,x_6, x_7	Q_2'''	**select**	x_4, x_6, x_7
	from	Person x_1,		**from**	Man_Made_Object x_3,
		x_1.carried_out.produced x_3,			x_3.produced_by.carried_out_by x_1,
		x_1.has_name x_2,			x_1.has_name x_2,
		x_3.has_title x_4,			x_3.has_title x_4,
		x_3.exposed_in x_5,			x_3.exposed_in x_5,
		x_5.museumName x_6,			x_5.museumName x_6,
		x_5.city x_7,			x_5.city x_7,
	where	x_2 = "Van Gogh"		**where**	x_2 = "Van Gogh"

Let us give a short discussion on the presence of *inverse roles* in the ontology :

- First they allow the user to formulate queries in a more natural way. The user can use any concept to start her query and follow roles and attributes appropriately to express the necessary constraints or to request the information that she looks for. For example, suppose that a user looks for *"the names of museums that expose the creations of Picasso"*. The user can start by concept Museum, and then follow the paths that lead her to the artifacts, and then the persons that created the latter. Without inverse roles, she might have to formulate the previous query by starting from concept Person, follow the path that leads to objects, and from there find the museums and the names thereof.

- Second, inverse rules allow us to use only *child/descendant* axes in the source paths of the mapping rules. In this case, the XML source is seen as a tree with a designated root, and the person in charge of specifying the mapping rules, is able to navigate downwards in order to map the XML structure to the ontology. If inverse roles were not present in the ontology, one should navigate in the XML tree until she finds an element to which she could map to a concept, and then move in any direction in the XML source to map the remaining elements.

Conclusions To conclude the discussion on query evaluation, a query q must be evaluated over each of the sources published in $STyX$. If the source does not provide answers for all variables in the query, then the part of the query that the source can answer is evaluated, and we look further for the missing parts of the answer from the other sources. This is a recursive process and stops when all sources have been examined. The partial results are then *joined* at the mediator site. When the query is evaluated against all sources and answers are found, *fusion* is performed on these results. Fusion is applied to the results to 'put together' information associated with an information entity which is found in different objects. The operations of *join* and *fusion* are formally presented in Section 4.2.3.

In the following we present the *$STyX$ data model* (Section 4.2) and the *mapping language* (Section 4.3). Section 4.5 discusses the $STyX$ query language. The *query rewriting* and *decomposition* algorithms are presented in Sections 4.6.1 and 4.6.2 and the *query execution plans* generator algorithm in Section 4.6.3.

4.2 $STyX$ Ontology

A $STyX$ ontology can be seen as an *object-oriented* schema. We define in this section our *ontology model.*

4.2.1 Ontology Data Model

We distinguish between *values* and *objects*. Let us now define the symbols to which we refer to during the presentation of our data model.
Let **Integer**, **String**, **Float**, **Real** denote the *atomic type names*[4]. Next we define the following sets whose domains are pairwise *disjoint* :

- The set **V** of atomic values is the union of the domains of the four atomic types **Integer**, **String**, **Real**, **Float**.
- The set **C** of *concept* symbols.
- The set **A** of *attribute* symbols.
- The set **R** of *role* symbols.

As in traditional object oriented models, *inheritance* allows us to model commonality of structures between concepts. We define a *concept hierarchy* that consists of the following two components : a *finite set of concept names*, and a *subclass relationship*.

Definition 4.2.1 (Concept Hierarchy) *A concept hierarchy is a pair* $H = (C, isa)$ *where* C *is a finite set of concept names in* **C**, *and isa defines a partial order on* C.

Definition 4.2.2 *Let* C *be a set of concepts,* $C \subseteq \mathbf{C}$. C *is upwards closed w.r.t. isa, if* $\forall c \in C$ *all its superconcepts are in* C.

Definition 4.2.3 (Ontology) *An ontology* $\mathcal{O}$ *is a 7-tuple* $\mathcal{O} = (C, V, R, A, source, target, isa)$ *where :*

- $C = \{c_1, c_2, \ldots c_n\}$ *is a set of concepts, subset of* **C**;
- V *is a set of atomic types* $\{\mathbf{Integer}, \mathbf{Real}, \mathbf{Float}, \mathbf{String}\}$;
- $R = \{r_1, r_2, \ldots r_n\}$ *is a set of roles, subset of* **R**;
- $A = \{a_1, a_2, \ldots a_n\}$ *is a set of attributes, subset of* **A**;
- *source is a function that maps a* role $r \in R$ *or an* attribute $a \in A$ *to its domain in* C *(source* : $R \cup A \rightarrow C$*);*
- *target is a function that maps a* role $r \in R$ *to its target in* C *and an attribute* $a \in A$ *to its domain in* V *(target* : $R \cup A \rightarrow C \cup V$*).*
- *the pair* (isa, C) *defines a* concept hierarchy.

The semantics of an ontology is defined by the databases that *conform to it.* A database 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*. Objects are also related to values by instances of attributes in A, which also satisfy the typing constraints implied by *source* and *target*.

As mentioned previously our ontology is a symmetric one : for each role $r \in R$, we define its *inverse*, denoted by $r^- \in R$ where $source(r^-) = target(r)$ and $target(r^-) = source(r)$. Roles and

[4]The atomic types are those defined in [26].

attributes are multi-valued and optional, i.e. any instance of concept $source(r)$ can be related to zero or more instances of concept (atomic type) $target(r)$ by role (attribute) r (a).

isa defines a hierarchy in $\mathcal{O}$, it carries subset semantics and supports role and attribute 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 (attributes) defined on concept c' are also defined in c. Note that two concepts that are not *isa* related are *not necessarily disjoint.*

In the following we define what is a database for an ontology $\mathcal{O}$.

Definition 4.2.4 (Ontology Database) *Given an ontology* $\mathcal{O} = (C, V, R, A, source, target, isa)$, *a* database $\mathcal{D}$ *is a labeled directed graph,* $\mathcal{D} = (N, \mathcal{V}, E, \lambda_N, \lambda_E, \psi)$ *where :*

- N *is the set of objects (nodes in the graph);*
- $\mathcal{V}$ *is the set of values (values in* V*, nodes in the graph);*
- E *is the set of edges;*
- λ_N *is a labeling function that labels each object* $n \in N$ *with a set of concepts,* $\lambda_N : N \rightarrow 2^C$ *(if* c *belongs in* $\lambda_N(o)$*, then* o *is called an instance of* c*);*
- λ_E *is a labeling function that labels each edge in the graph with a role* $r \in R$ *or an attribute* $a \in A$, $\lambda_E : E \rightarrow R \cup A$.
- ψ *is an incidence function that associates each edge to its domain in* N *and target in* $N \cup \mathcal{V}$, $\psi : E \rightarrow N \times (N \cup \mathcal{V})$;

The constraints that must be satisfied by a database $\mathcal{D}$ *of* $\mathcal{O}$ *are the following :*

- $\forall o \in N$, $\lambda_N(o)$ *is upwards closed with respect to isa : if an object* o *is an instance of concept* c *then it is an instance of all superconcepts of* c*;*
- $\forall e \in E$, *let* $\psi(e) = (o, o')$ *where* o, o' *are objects in* N *and* $\lambda_E(e) = r$ *where* r *is a role in* R. *Then, there must exist concepts* c, c' *where* o *is an instance of* c *and* o' *is an instance of* c', *such that* $source(r) = c$ *and* $target(r) = c'$.
- $\forall e \in E$, *let* $\psi(e) = (o, v)$ *where* o *is an object in* N *and* v *is a value in* $\mathcal{V}$. *Let* $\lambda_E(e) = a$ *where* a *is an attribute in* A. *Then, there must exist a concept* c, *where* o *is an instance of* c, *and an atomic type* $d \in V$, *where* v *is in the domain of* d, *such that* $source(a) = c$ *and* $target(a) = d$.

4.2.2 Ontology Paths

In the *local as view* approach we assume a *hypothetical* global database that contains all the information of interest to the user in a given domain of interest. Each of the data sources is described as a *materialized view* of the global database and contains *only part* of the data. For example, a source might contain information about some entity types (concepts) but not about others. For a given entity type, it might contain only some instances of this type. And for a given instance it might contain only a subset of the roles/attributes defined on its entity type.

As illustrated by the examples in Section 4.1.3, a source is described by a set of *mapping rules.* Consider source *http://www.paintings.com* whose DTD is illustrated in Figure 4.1 and the set of mapping rules by which it is published in the ST_YX mediator, shown in Figure 4.6. This source returns a subset of the instances of concept Person. It returns their names (attribute *has_name*) and

their year of birth (path *born.took_place_at.year*). The source does not contain any information about the day and month of a person's birth date, neither her place of birth : *it contains only part of the relevant data for a person.* Moreover, the source describes objects that are created by persons. But, the source does not describe the activities (instances of concept Activity) which are carried out by persons, and by which these objects are produced : *the structure of the source is different from the structure of the global schema.*

In order to describe the source structure in terms of the ontology, we introduce a simple *path-based ontology language.* This language allows one to create new concepts and roles to describe source structures which cannot be directly mapped to the ontology. This need arises from the nature of the ontology which is defined *independently* of the structure of the sources and contains concepts that are essential in the domain of discourse but are not necessarily represented in a source.

In the following, we define our *path-based ontology language.* It is based on the notions of *role*, *attribute* and *concept paths* which allow one to define new structures at the mediator level. We also show in detail how the global schema is extended with the notion of *keys.*

Role/Attribute Paths and Derived Roles/Attributes

Definition 4.2.5 (Role Paths) *A* role path *of length n ($n \geq 1$) is a sequence $p = r_1 \ldots r_n$, where $\forall i \in [1,n]$, r_i are roles such that either $target(r_i)$ isa $source(r_{i+1})$ or $target(r_i) = source(r_{i+1})$ for $1 \leq i < n$. The source and target of p are defined by the source and target of its extremities : $source(p) = source(r_1)$ and $target(p) = target(r_n)$.*

Definition 4.2.6 (Inverse Paths) *Let $p = r_1.r_2 \ldots r_n$ be a role path. Its inverse denoted by p^- is defined as $p^- = r_n^-.r_{n-1}^- \ldots r_1^-$ where r_j^- is the inverse role of r_j.*

Definition 4.2.7 (Attribute Paths) *An* attribute path *of length n ($n \geq 1$) is a sequence $p = r_1 \ldots r_n$, where $\forall i \in [1, n-1]$, r_i are roles such that either $target(r_i)$ isa $source(r_{i+1})$ or $target(r_i) = source(r_{i+1})$ and r_n is an attribute. The source and target of p are defined by the source and target of its extremities : $source(p) = source(r_1)$ and $target(p) = target(r_n)$.*

Each role of R is a role path of length one; each attribute of A is an attribute path of length one. We also define *composition* of role paths and attribute paths as follows.

Definition 4.2.8 (Composition of Paths) *Let $p_1 = \alpha_1.\alpha_2.\ldots.\alpha_n$ be a role path and $p_2 = \beta_1.\beta_2.\ldots.\beta_k$ be a role path or an attribute path. The composition $p = p_1 \circ p_2$ is a role path (or attribute path if p_2 is an attribute path) $p = \alpha_1.\alpha_2 \ldots \alpha_n.\beta_1.\beta_2.\ldots.\beta_k$ of length $n+k$, if either $target(p_1)$ isa $source(p_2)$ or $target(p_1) = source(p_2)$. For p, $source(p) = source(p_1)$ and $target(p) = target(p_2)$.*

Definition 4.2.9 (Derived Roles/Attributes) *A* role/attribute path *$p = r_1.r_2.\ldots.r_n$ of length > 1 defines a* derived role/attribute *denoted by $role(p)/att(p)$. Instances of a derived role/attribute connect instances of $source(r_1)$ and instances of $target(r_n)$.*

For example, role path r = *carried_out.produced* defines a *derived role*, $role$(*carried_out.produced*), between concept Actor ($source(r)$) and concept Man_Made_Object ($target(r)$). Instances of this role connect instances of concept Actor and instances of concept Man_Made_Object.

We denote by $\mathcal{RP}_\mathcal{O}$ (or simply $\mathcal{RP}$ if $\mathcal{O}$ is known), the set of all role paths in $\mathcal{O}$ and by $\mathcal{AP}_\mathcal{O}$ (or simply $\mathcal{AP}$ if $\mathcal{O}$ is known) the set of all attribute paths of $\mathcal{O}$. $\mathcal{RP}_\mathcal{O}$ and $\mathcal{AP}_\mathcal{O}$ are *infinite* if $\mathcal{O}$ is cyclic.

Concept Paths and Derived Concepts

Definition 4.2.10 (Concept paths) *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 either $c = source(r)$ or c isa $source(r)$.*

The length of a concept path p is 0 if $p = c$ and the length of the role path r if $p = c.r$. If $p = c$ then $source(p) = c$ and $target(p) = c$. If $p = c.r$ then $source(p) = c$ and $target(p) = target(r)$.

Definition 4.2.11 (Composition of Concept Paths and Role Paths) *Let $p_1 = c.r$ be a concept path and $p_2 = r_1.r_2 \ldots r_n$ be a role path. The composition of p_1 and p_2 denoted by $p = p_1 \circ p_2$ is a concept path $p = c.r.r_1.r_2 \ldots r_n$, if either $target(r)$ isa $source(r_1)$ or $target(r) = source(r_1)$.*

Definition 4.2.12 (Derived Concepts) *A concept path $p = c.r$ where c is a concept and r is is a role path of length ≥ 1 defines a derived concept denoted by $conc(p)$. Instances of this concept are* "the instances of $target(p)$ that can be reached from instances of $source(p)$ by following the roles in r in order of their appearance in r."

Consider concept path $p =$ Person.*carried_out.produced*. It defines the derived concept *conc*(*Person.carried_out.produced*) that stands for the instances of concept Man_Made_Object (*target*(p)) reached by instances of concept Person following the roles in path *carried_out.produced*.

In the following $\mathcal{CP}_{\mathcal{O}}$ (or simply $\mathcal{CP}$ if $\mathcal{O}$ is known) denotes the set of concept paths of ontology $\mathcal{O}$. If $\mathcal{O}$ is cyclic then $\mathcal{CP}$ is infinite.

Definition 4.2.13 (Suffixes of Concept Paths) *Let $p = c.r_1.r_2.\ldots.r_n$ be a concept path. A concept path p' is a suffix of p iff :*

- *$p' = c'$ where $c' = target(r_n)$ or $target(r_n)$ isa c';*
- *$p' = c'.r_k.r_{k+1} \ldots r_n$ $(1 < k \leq n)$ and $target(r_{k-1})$ isa c' or $c' = target(r_{k-1})$;*
- *$p' = c'.r_1.r_2.\ldots.r_n$ where c isa c'.*

Concept path $p =$ Person.*carried_out.produced* has three suffixes: (i) Actor.*carried_out.produced*, (ii) Activity.*produced* and (iii) Man_Made_Object. The first suffix is obtained by replacing Person by its superconcept Actor. The second suffix is obtained by removing role *carried_out* from the path and adding the concept Activity in the beginning (Activity=*target*(*carried_out*)). Concept path p' = Activity.*produced* defines a derived concept that stands for the instances of Man_Made_Object (*target*(p')) that can be reached from instances of concept Activity by following role *produced*. If we remove the last role (i.e. *produced*) and replace it by its *target* i.e. Man_Made_Object, then we obtain the third suffix.

Concepts, derived concepts and their suffixes are related by *derived isa*. If p' is a suffix of concept path p, then the derived concept *conc*(p) is a subconcept of the derived concept *conc*(p'). That is for a given database, the extent of *conc*(p) is a subset of the extent of *conc*(p'). We define in the following more formally the notion of *derived isa* between derived concepts.

4.2.3 Keys

As illustrated by the examples in Section 4.1.4, *keys* are essential in our context for *identifying* objects (i.e. XML fragments) in order to perform *identity-based join* and *fusion*.

The Tsimmis [170] and YAT [49] projects concerning semi-structured data integration, deal with the problem of *object fusion*. In their context, object fusion is similar to our *identity-based* join. To

perform fusion, objects in these approaches, are identified by *semantic keys*. For example, in the Tsimmis [170] project, the identity of an object is determined by the value of one of its *subobjects*. A similar approach is followed in [54] where Skolem functions are applied to the value of a specific attribute of an object to obtain the object's identifier. The Agora system [149] considers *persistent object identifiers* to perform fusion. Nevertheless, persistent object identifiers are not a solution in the Web context where each source handles *independently* of the others its identifiers and is not willing or able to have some common agreement on how these identifiers are managed.

In our context, we introduce two types of keys to identify objects :

- *local keys* are provided locally by the XML sources without taking into consideration the other sources. In this category fall the XML attribute of type `ID` and the *position* of an XML fragment in an XML document. Fragments in an XML document can be identified by the value of an attribute of type `ID` (if such an attribute is defined), or by its *unique position* in the original document. It is evident that local keys cannot be used to identify fragments coming from different documents.

- *global keys* are *semantic keys* which are defined in *concepts* at the *ontology level* and are used to identify objects that originate from different or the same source.

Local Keys XML fragments are not associated always with an attribute of type `ID`. The position of fragments, when the document is considered as an ordered tree, can always be used to identify them.

For convenience, an object represented by an XML fragment, is labeled with its position in the XML document and the value of the `ID` attribute using functions θ[5] and τ[6] respectively. The source might also generate local identifiers (e.g. using the function `generate-id()` provided by XSLT [82]). An object can have more than one values for θ, τ.

To obtain the value for an attribute of type `ID` of an XML fragment we must be aware of the XML DTD of the source (i.e. the name of the attribute). Obtaining the position of the XML fragment is more complex though since currently there is no way to obtain this information by some function of XPath. Nevertheless, these features are included in our model and if the source returns this information it can be used for query answering.

Global Keys Even for XML fragments originating from the same document, local keys cannot always be used to identify them. Let us illustrate this by an example. Consider the XML DTD illustrated below which describes artists (element `Artist`). Each artist is associated with only one artifact (element `Artifact`). This means that an artist with more than one artifacts appears more than once in the XML document. An artist is associated with a name (attribute `Name`) and an artifact is associated with its title (attribute `Title`).

```
<!ELEMENT Artist (Artifact)>
<!ATTLIST Artist Name #CDATA #REQUIRED>
<!ELEMENT Artifact EMPTY>
<!ATTLIST Artifact Title #CDATA #REQUIRED>
```

Consider the valid document for this XML DTD illustrated below.

[5] θ is the initial of the Greek word $\theta\acute{\epsilon}\sigma\eta$ which means position in Greek.
[6] τ is the initial of the Greek word $\tau\alpha\upsilon\tau\acute{o}\tau\eta\tau\alpha$ which means identity in Greek.

```
<Artist Name='Van Gogh' Year = '1800'>
  <Artifact Title='Starry Night'/>
</Artist>
<Artist Name='Van Gogh' Year='1800'>
  <Artifact='Vincent's Room at Arles'/>
</Artist>
```

We see from the example document that two different XML fragments correspond to the same real world entity (i.e. the same person). These fragments do not come with `ID` attributes, and have different positions in the document. Moreover, we cannot define the attribute `Name` as an attribute of type `ID` since the XML standard [60] does not allow two different fragments to have the same value for such an attribute.

A *global key is defined at the ontology level* and can be used to identify objects independently of their origin. A *global key* for a concept is a set of *attribute paths* originating from it. A concept can be associated with zero, one or more keys. These keys are defined at the mediator level, independently of the sources.

Definition 4.2.14 (Global Key) *Let $\mathcal{AP}$ be the set of attribute paths for an ontology $\mathcal{O}$. A global key k for a concept $c \in C$ is a set of attribute paths, $k = \{a_1, a_2, \ldots a_n\}$, $k \subseteq \mathcal{AP}$ such that $\forall\ a_i,\ source(a_i) = c$.*

Definition 4.2.15 (Ontology with Global Keys) *An ontology extended with global keys is a pair $(\mathcal{O}, K)$ where $\mathcal{O}$ is as defined previously and K is a function which associates with each concept in C, a possibly empty collection of finite sets of keys ($K : C \rightarrow 2^{2^{\{\mathcal{AP}\}}}$).*

From this point on, the term ontology denotes an ontology with keys.

Example 4.2.1 *Consider concept* Person. *A person is identified by her name (value of attribute* has_name*) and her year of birth (value of attribute path* born.took_place_in.year*) :*

$$K(\mathsf{Person}) = \{\{has\_name,\ born.took\_place\_in.year\}\}$$

An instance of concept Place, *is identified by its name.*

$$K(\mathsf{Place}) = \{\{placeName\}\}$$

An instance of concept Man_Made_Object *is identified by its title and by its year of creation, or by its museum identifier :*

$$K(\mathsf{Man\_Made\_Object}) = \{\{has\_title,\ created.took\_place\_in.year\},\ \{museumIdentifier\}\}$$

When a key is defined for a concept, then it is inherited by all its subconcepts. Moreover, we can define more keys for the latter, but it is not possible to *refine* the keys defined in the former. For example, consider concept Person and its subconcept Painter. The key $k_1 = \{has\_name\}$ is defined for the former, and is inherited to the latter. It is useless to define in concept Painter key $k_2 = \{has\_name, born.took\_place\_in.year\}$ since there is no point in testing object identity using k_2 if the objects agree on their values for k_1.

Definition 4.2.16 *Let c, c' be concepts in $\mathcal{O}$, where c' isa c. Then $K(c) \subseteq K(c')$ where $\forall\ k_i, k_j$ such that $k_j \in K(c)$, $k_i \in K(c') - K(c)$, then $k_i \cap k_j = \emptyset$.*

We must note here that an object can have multiple values for a global key. For example, in a source, a person can have one or more names. This is a realistic assumption based on (i) the semi-structured nature of XML and (ii) on the fact that XML DTDs do not come with keys! (except for the XML `ID` attribute which is a local key). This approach is also undertaken in [32, 84] for defining keys for XML.

Definition 4.2.17 *Let c be a concept in the ontology, and k be a key for c, where* $k = \{a_1, a_2, \ldots a_n\}$ *and the* a_i*'s are attribute paths. Let o be an object in* $\mathcal{D}$*, and* $c \in \lambda_N(o)$*. Let* $\{e_1, e_2, \ldots e_n\}$ *be edges in E such that* $\forall e_i, \psi(e_i) = (o, u_i)$ *and* $\lambda_E(e_i) = a_i$*. Then, a value for key k and for object o is denoted by the record* $[a_1 : u_1, a_2 : u_2, \ldots a_n : u_n]$.

Function κ *associates each triple* (o, c, k)*, where o is an object, c is a concept where* $c \in \lambda_N(o)$ *and* $k \in K(c)$*, with a possibly empty set of records of the form* $[a_1 : u_1, \ldots a_n : u_n]$. $\kappa : (N \times C \times \mathcal{K}) \rightarrow 2^{[a_1:u_1,\ldots a_n:u_n]}$ *where* $\mathcal{K}$ *is the set of global keys in the ontology.*

Object identity Given the previous definitions for global and local keys we define when two objects in a database $\mathcal{D}$ are *identical*. We distinguish between identity based on *local* and *global* keys : two objects that have the same value for at least one of their local keys are called *locally identical* (they correspond to the same XML fragment). Two objects that have the same value for *at least one* of their global keys are called *globally identical*.

Definition 4.2.18 *(***Local Identity***) Let o,* o' *be two objects. We say that o and* o' *are* locally identical *if either* $\tau(o) \cap \tau(o') \neq \emptyset$ *or* $\theta(o) \cap \theta(o') \neq \emptyset$*. In the first case* $o =_\tau o'$*, and in the second* $o =_\theta o'$.

In the case of global keys, in order to decide whether two objects are *globally identical* we require that (i) are instances of the *same concept* and (ii) they have *at least one common value* for a key for this concept.

Definition 4.2.19 (Global Identity) *Let* $(\mathcal{O}, K)$ *be an ontology with keys. Let* $\mathcal{D}$ *be a database that conforms to* $\mathcal{O}$*. Two objects o,* o' *are globally identical* ($o =_\kappa o'$) *if :* $\exists c \in \lambda_N(o) \cap \lambda_N(o')$ *and* $\exists\ k \in K(c)$ *and* $\kappa(o, k, c) \cap \kappa(o', k, c) \neq \emptyset$.

From this point on, we refer to two objects as identical ($o = o'$) if they are either locally or globally identical.

Definition 4.2.20 (Legal Database) *Let* $(\mathcal{O}, K)$ *be an ontology with keys and* $\mathcal{D}$ *be a database that conforms to* $\mathcal{O}$*.* $\mathcal{D}$ *is a legal database for* $\mathcal{O}$ *if there do not exist two identical objects o and* o'.

4.2.4 Derived Ontology

The augmentation of a given ontology with the derived roles, attributes and concepts gives a *derived ontology*. It is significant for the integration, since it provides an interpretation for the *mapping rules* by which sources are published in the ST_YX mediator, hence for query processing as we illustrate later.

Given the simple *path-based ontology language* described earlier, we define the notion of *derived ontology*. First, the notion of *derived concept hierarchy* is formally defined.

Definition 4.2.21 (Derived Concept Hierarchy) *Let $\mathcal{O}$ be an ontology as previously defined. A derived concept hierarchy $H^{\mathcal{X}} = (C^{\mathcal{X}}, isa^{\mathcal{X}})$ where $C^{\mathcal{X}}$ is the set of derived concepts defined by the set of concept paths $\mathcal{CP}$, and $isa^{\mathcal{X}}$ is a partial order on $C^{\mathcal{X}}$ (transitive, antisymmetric and reflexive). $isa^{\mathcal{X}}$ is defined as follows :*

- *for all concepts c, c' in $\mathcal{O}$ such that c is a subconcept of c' (c isa c') then c is a subconcept of c' in $\mathcal{O}^{\mathcal{X}}$: $\forall c, c' \in C$ if c isa c', then c $isa^{\mathcal{X}}$ c';*
- *the derived concept defined by some concept path p in $\mathcal{CP}$ is a subconcept in $\mathcal{O}^{\mathcal{X}}$ of the derived concepts defined by its suffixes : $\forall p \in \mathcal{CP}$, and for all suffixes p' of p, then conc(p) $isa^{\mathcal{X}}$ conc(p').*

The derived ontology $\mathcal{O}^{\mathcal{X}}$ contains (i) the derived concepts defined by concept paths in $\mathcal{CP}$, (ii) the derived roles defined by the role paths in $\mathcal{RP}$ and (iii) the derived roles defined by the attribute paths in $\mathcal{AP}$.

Definition 4.2.22 (Derived Ontology) *Let $\mathcal{O} = (C, V, R, A, source, target, isa)$ be an ontology. Let $\mathcal{RP}$ be the set of role paths, $\mathcal{AP}$ be the set of attribute paths, and $\mathcal{CP}$ be the set of concept paths in $\mathcal{O}$. The* derived *ontology $\mathcal{O}^{\mathcal{X}} = (C^{\mathcal{X}}, V, R^{\mathcal{X}}, A^{\mathcal{X}}, source^{\mathcal{X}}, target^{\mathcal{X}}, isa^{\mathcal{X}})$ is defined as follows :*

- *$C^{\mathcal{X}}$ is the set of derived concepts defined by the concept paths in $\mathcal{CP}$;*
- *$R^{\mathcal{X}}$ is the set of derived roles defined by the role paths in $\mathcal{RP}$;*
- *$A^{\mathcal{X}}$ is the set of derived attributes defined by the attribute paths in $\mathcal{AP}$[7];*
- *V is the set of atomic type names;*
- *$source^{\mathcal{X}}$ is a function that maps a role $r \in R^{\mathcal{X}}$ or an attribute $a \in A^{\mathcal{X}}$ to its domain in C ($source^{\mathcal{X}} : R^{\mathcal{X}} \cup A^{\mathcal{X}} \rightarrow C$)[8];*
- *$target^{\mathcal{X}}$ is a function that maps a role $r \in R^{\mathcal{X}}$ to its target in C and an attribute $a \in A^{\mathcal{X}}$ to its target in V ($target^{\mathcal{X}} : R^{\mathcal{X}} \cup A^{\mathcal{X}} \rightarrow C \cup V$)[9];*
- *the pair $(C^{\mathcal{X}}, isa^{\mathcal{X}})$ defines a derived concept hierarchy;*

The keys defined for concepts in $\mathcal{O}$ are defined for concepts in $\mathcal{O}^{\mathcal{X}}$. A key defined for a concept c is also defined for all its subconcepts c' where c' $isa^{\mathcal{X}}$ c.

Definition 4.2.23 *(***Derived Database***) Given an ontology $\mathcal{O}$ and a database $\mathcal{D} = (N, \mathcal{V}, E, \lambda_N, \lambda_E, \psi)$ of $\mathcal{O}$, the* derived database *$\mathcal{D}^{\mathcal{X}} = (N^{\mathcal{X}}, \mathcal{V}, E^{\mathcal{X}}, \lambda_N^{\mathcal{X}}, \lambda_E^{\mathcal{X}}, \psi)$, derived from $\mathcal{D}$, instance of $\mathcal{O}^{\mathcal{X}}$, is constructed as follows :*

- *the set of objects in $\mathcal{D}^{\mathcal{X}}$ is the same as the set of objects in $\mathcal{D}$: $N^{\mathcal{X}} = N$;*
- *the set of edges in $\mathcal{D}^{\mathcal{X}}$ is initialized by the set of edges E in $\mathcal{D}$. $E^{\mathcal{X}}$ is extended by creating* edge paths *as follows : for all edges $e_1, e_2, \ldots e_n \in E^{\mathcal{X}}$, with $\lambda_E^{\mathcal{X}}(e_i) = r_i$ (where r_i is a role for $i \leq n$ or an attribute for $i = n$) if*

[7] By the previous definitions $C \subseteq \mathcal{CP}$, $R \subseteq \mathcal{RP}$, $A \subseteq \mathcal{AP}$.

[8] Here we note that $source^{\mathcal{X}}$ is restricted to map a role/attribute to C. We could map it to $C^{\mathcal{X}}$ but this information is not used during query processing.

[9] Here we note that $target^{\mathcal{X}}$ is restricted to map a role/attribute to $C \cup V$. We could map it to $C^{\mathcal{X}} \cup V$ but this information is not used during query processing.

1. $\exists\, o_1, o_2, \ldots o_{n+1}$ *such that* $\forall i, \psi(e_i) = (o_i, o_{i+1})$ *i.e.* $e_1, e_2, \ldots e_n$ *define a path in* $E^{\mathcal{X}}$ *and*
2. $\forall i \in [1, n-1]$, *either* $target^{\mathcal{X}}(r_i)$ $isa^{\mathcal{X}}$ $source^{\mathcal{X}}(r_{i+1})$ *or* $target^{\mathcal{X}}(r_i) = source^{\mathcal{X}}(r_{i+1})$ *and in particular* $source^{\mathcal{X}}(r_i) \in \lambda_N^{\mathcal{X}}(o_i)$ *for* $i \in [1, n]$;

then $\exists\, e' \in E^{\mathcal{X}}$ *such that* $\psi(e') = (o_1, o_{n+1})$ *and* $\lambda_E^{\mathcal{X}}(e') = r_1.r_2.\ldots.r_n$.

- *the labels of objects in* $\mathcal{D}^{\mathcal{X}}$ *are extended as follows : if there exist an object in* $\mathcal{D}^{\mathcal{X}}$ *reached by a concept path, then the derived concept defined by the latter is added in the set of labels of the object. For each* $o \in N^{\mathcal{X}}$ *and for each* $e \in E^{\mathcal{X}}$ *where* $\lambda_E^{\mathcal{X}}(e) = p$ *(*p *is a derived role) and* $\psi(e) = (o', o)$ *such that* o' *is an object in* $N^{\mathcal{X}}$ *and* $c \in \lambda_N^{\mathcal{X}}(o')$, $c = source^{\mathcal{X}}(p)$, *then* $conc(c.p) \in \lambda_N^{\mathcal{X}}(o)$. $\lambda_N^{\mathcal{X}}(o)$ *is then closed upwards.*

Notice that in the definition of $\mathcal{D}$ and $\mathcal{D}^{\mathcal{X}}$ (databases for $\mathcal{O}$ and $\mathcal{O}^{\mathcal{X}}$ respectively) we introduce two different functions λ_N and $\lambda_N^{\mathcal{X}}$ to label objects. λ_N is a labeling function that associates with each object in N, a set of *concepts* in $\mathcal{O}$. $\lambda_N^{\mathcal{X}}$ is a function that associates with each object in $N^{\mathcal{X}}$, a set of derived concepts (defined by concept paths) in $\mathcal{O}^{\mathcal{X}}$. The same holds for functions λ_E and $\lambda_E^{\mathcal{X}}$. λ_E labels an edge in $\mathcal{D}$ with a role or an attribute of $\mathcal{O}$. $\lambda_E^{\mathcal{X}}$ labels an edge in $\mathcal{D}^{\mathcal{X}}$ with a derived role or a derived attribute (defined by some role, attribute path) of $\mathcal{O}^{\mathcal{X}}$.

An example of a database and its derived is illustrated in Figure 4.11. Notice that in the derived database, no new objects are created. Nevertheless, derived concepts are added in the object labels. For example, object o_2 is labeled with concepts[10] $conc(C_2)$ and $conc(C_4)$ (as in the original ontology) and the derived concept $conc(C_1.r_1)$. Object o_3 is labeled with concept $conc(C_3)$ and with the derived concepts $conc(C_1.r_1.r_2)$, $conc(C_2.r_2)$ and $conc(C_4.r_2)$. The latter is defined by the *suffix* $C_4.r_2$ of concept path $C_2.r_2$. Edge e_3 is created. It connects objects o_1 and o_3 and its label is $role(r_1.r_2)$.

We have to note here that the key value of an object o in $\mathcal{D}^{\mathcal{X}}$ is defined by the database itself. For example, consider that o is an instance of concept c. Let $k = \{a_1, a_2, \ldots a_n\}$ be a key for concept c, where the a_i's are attribute paths. Let $\{e_1, e_2, \ldots e_n\}$ be a set of outgoing edges of o, where $\forall\, e_i, \lambda_E^{\mathcal{X}}(e_i) = a_i$, and $\forall\, i, \psi(e_i) = (o, v_i)$. Then, the record $[a_1 : v_1, a_2 : v_2, \ldots a_n : v_n]$ is added in the set of key values $\kappa(o, c, k)$.

4.2.5 Identity-based Join and Fusion

In this section we define the operations of *identity-based join* and *fusion*. The former is a *binary* operation and is applied on two objects. It returns a new object whose structure is the merge of the structure of the two objects. Fusion is a *unary* operation which is applied on a database of objects and returns a new database where all objects sharing some key value have been replaced by a single new object.

Join We define first the operations of *local* and *global join*. The former is applied to objects that are *locally identical* and the latter to objects which are *globally identical*. Then we define the operation of *LG-join* in terms of the above.

Definition 4.2.24 *(***Local Join***) Let* o_1, o_2 *be two objects.* o_1 *and* o_2 *can be* locally joined *if they are locally identical. We distinguish between the local join based on the* **ID** *XML attribute and the one based on the position of the XML fragments (representing the objects) in the XML document. The first is denoted by* $o_1 \bowtie_\tau o_2$ *and the second by* $o_1 \bowtie_\theta o_2$.

[10] In Figure 4.11 the concept paths and role paths are shown.

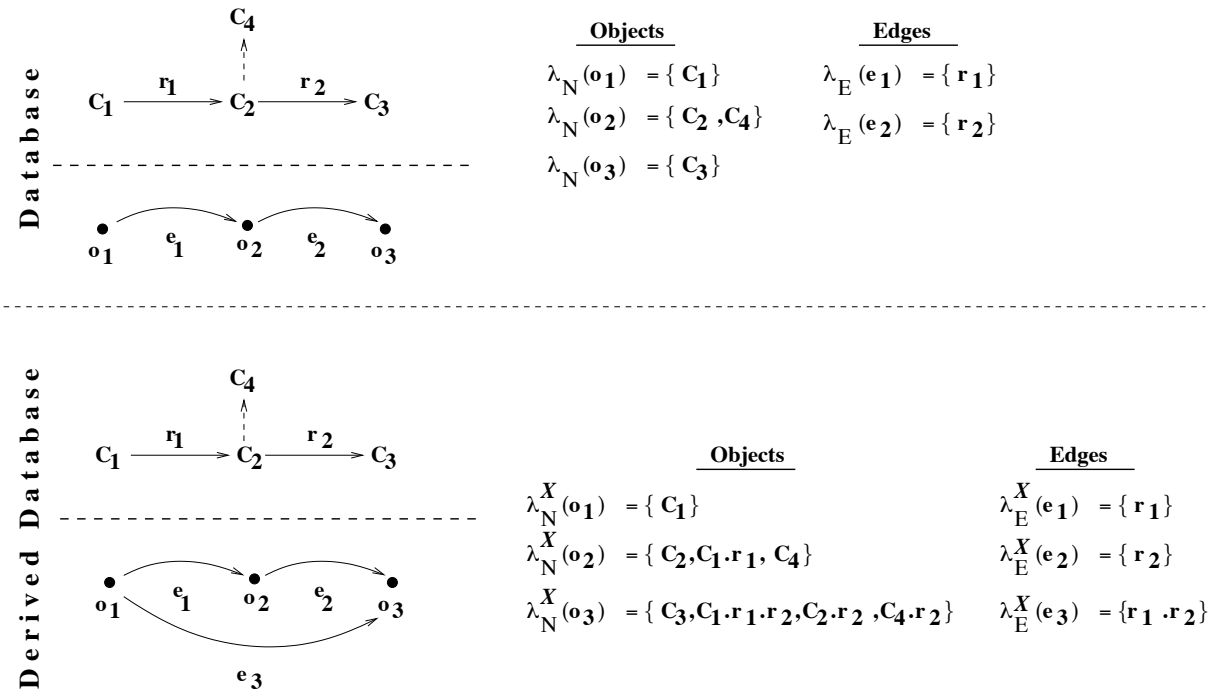

Figure 4.11: A database $\mathcal{D}$ and the derived database $\mathcal{D}^{\mathcal{X}}$.

Definition 4.2.25 *(**Global Join**) Let o_1, o_2 be two objects. o_1 and o_2 can be* globally joined *if they are globally identical. Global join is denoted by $o_1 \bowtie_\kappa o_2$.*

Definition 4.2.26 *(**LG-Join**) The $LG-Join$ between objects o_1 and o_2 is defined as follows :*

$$o_1 \bowtie_{LG} o_2 \quad = \quad \begin{cases} o_1 \bowtie_\tau o_2 & \text{if } o_1 =_\tau o_2 \\ o_1 \bowtie_\theta o_2 & \text{if } o_1 =_\theta o_2 \\ o_1 \bowtie_\kappa o_2 & \text{if } o_1 =_\kappa o_2 \end{cases}$$

Definition 4.2.27 *Let o_1 and o_2 be two identical objects in a database $\mathcal{D}^{\mathcal{X}}$, instance of $\mathcal{O}^{\mathcal{X}}$. The result of the $LG-join$ between o_1 and o_2 is a new object o ($o = o_1 \bowtie_{LG} o_2$) such that :*

1. *o is an instance of the concepts whose instances are o_1 and o_2 : $\lambda^{\mathcal{X}}_N(o) = \lambda^{\mathcal{X}}_N(o_1) \cup \lambda^{\mathcal{X}}_N(o_2)$;*
2. *all outgoing edges of o_1 and o_2 become outgoing edges of o : $\forall e \in E^{\mathcal{X}}$ such that $\psi(e) = (o', x)$ where o' is one of o_1, o_2 and x is a value or an object, then $\psi(e) = (o, x)$;*
3. *all incoming edges of o_1 and o_2 become incoming edges of o : $\forall e \in E^{\mathcal{X}}$ where $\psi(e) = (x, o')$ such that o' is one of the o_1, o_2 and x is an object, then $\psi(e) = (x, o)$;*
4. *for all edges $e_1, e_2, \ldots e_n$ between object o and some object o' such for all pairs e_i, e_j $i \neq j$, $i, j \in [1, n]$, $\lambda^{\mathcal{X}}_E(e_i) = \lambda^{\mathcal{X}}_E(e_j)$ then merge $e_1, e_2, \ldots e_n$ into one edge;*

In the case of local and global joins, $\theta(o) = \theta(o_1) \cup \theta(o_2)$, $\tau(o) = \tau(o_1) \cup \tau(o_2)$. Global keys for o are derived from the new database. Let $c \in \lambda^{\mathcal{X}}_N(o)$ and $k \in K(c)$ where $k = \{a_1, a_2, \ldots a_n\}$. Let $e_1, e_2, \ldots e_n \in E^{\mathcal{X}}$ where $\forall e_i, \lambda^{\mathcal{X}}_E(e_i) = a_i, \psi(e_i) = (o, u_i)$, u_i is a value. Then $\kappa(o, c, k) = \kappa(o, c, k) \cup \{[a_1 : u_1, a_2 : u_2, \ldots a_n : u_n]\}$[11].

[11]It might be the case, that adding edges originating from the objects o_1, o_2 to o, we get values for a key for o.

1. Input: The database $\mathcal{D}^{\mathcal{X}}$;
2. Output: The database $\mathcal{D}^{\mathcal{X}}_{\Phi}$;
3. Algorithm: initialization : $\mathcal{D}^{\mathcal{X}}_{\Phi} = \mathcal{D}^{\mathcal{X}}$;
4. loop : for all objects o in $N^{\mathcal{X}}$ {
5. $S = partition(o, N^{\mathcal{X}} - \{o\})$;
6. /* S contains the objects in $N^{\mathcal{X}}$ which are identical to o */
7. if $S \neq \emptyset$ {
8. $o' = join(o, S)$
9. /* o' is the result of the $LG - join$ between o and the objects in S */
10. $N^{\mathcal{X}} = N^{\mathcal{X}} - S - \{o\} \cup \{o'\}$
11. /* update $N^{\mathcal{X}}$: remove the objects in S and o */
12. /* and add new object o' */
13. }
14. }
15. return database $\mathcal{D}^{\mathcal{X}}_{\Phi}$;

Figure 4.12: Fusion Algorithm

Fusion Let $\mathcal{D}^{\mathcal{X}}$ be a database, instance of $\mathcal{O}^{\mathcal{X}}$. The result of applying fusion on this database is a *new database*. It is obtained from the initial one by applying *recursively* $LG - join$ between all pairs of identical objects. If two objects are joined then they are replaced by the resulting object.

The fusion operator is denoted by Φ and the result database is $\mathcal{D}^{\mathcal{X}}_{\Phi} = \Phi(\mathcal{D}^{\mathcal{X}})$. The algorithm is illustrated in Figure 4.12.

The above algorithm makes use of the functions *partition* and *join*. The first accepts as input an object o and a set of objects A, and returns the set of objects in A which are identical to o. The second accepts as input an object and the set S of its identical objects and returns the object produced by applying recursively $LG - join$ between o and the objects in S.

Example 4.2.2 *Consider the database $\mathcal{D}^{\mathcal{X}}$ illustrated in the right part of Figure 4.13. The result of the fusion between objects o_2, o_3 is object o_{23} and the result of the fusion of o_5, o_6 is object o_{56}. The incoming edges of object o_{23} are labeled with roles r_1,r_2 and r_3 (the incoming edges of objects o_2 and o_3). Notice that there is only one edge labeled with r_3 from object o_1 : the edges labeled with r_3 originating from object o_1 and arriving in objects o_2 and o_3 are merged into a single edge since they have the same label. Objects o_2, o_3 and o_5, o_6 disappear and are replaced by objects o_{23} and o_{56}.*

4.3 Mapping Language

In this section we define the *ST$_Y$X mapping language*. It is based on establishing *mappings* between *XPath location paths* and *ontology paths*.

Definition 4.3.1 (Mapping Rule) *Let V be a set of variables, and U be a set of URLs. A* mapping rule *is an expression of the form $R : u/q$ as $v \rightarrow p$, where*

- *R is the rule's* label,

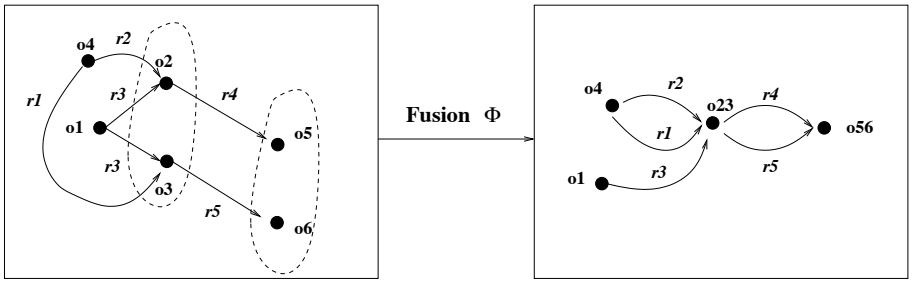

Figure 4.13: Object Fusion

operation *partition*
Input : object o and a set of objects S;
Output : a set of objects $S' \subseteq S$ which are identical to o;
Algorithm : initialization $S' = \emptyset$;
loop : for all objects $o' \in S$ {
/* *if* o' *is identical to* o, *add* o' *in* S' */
if $o' = o$, $S' = S' \cup \{o'\}$
}
return S';

operation *join*
Input : object o and a set S of identical objects of o;
Output : object o;
Algorithm : loop : for all objects $o' \in S$ {
/* *apply* $LG - join$ *between* o *and* o' */
$o = o \bowtie_{LG} o'$
}
return o;

Figure 4.14: Operations *partition* and *join*

- $u \in V \cup U$, *the rule's* root, *is either a variable or a URL,*
- q *is an XPath location path, called* source path *of the rule and denoted by* $sp(R)$,
- *as* v *is a variable declaration (*v *is called the bound variable of* R*) and*
- p *is an ontology path, denoted by* $op(R)$ *: more precisely,* p *is a role/attribute path if* u *is a variable and a concept path otherwise.*

Rule R is a *relative mapping rule* if its root is a variable v, and an *absolute mapping rule* otherwise.

Definition 4.3.2 (Reachability for Rules and Variables) *Given a set of mapping rules defined for a source* s*, and an ontology* $\mathcal{O}$*, we define* reachability *for rules and variables, as follows :*

- *each rule whose root is a URL is reachable;*
- *each variable bound in a reachable rule is reachable;*
- *each rule whose root variable is a reachable variable is reachable.*

Definition 4.3.3 (Mapping) *A mapping* M *for a source* s *and an ontology* $\mathcal{O}^{\mathcal{X}}$ *is a set of* mapping rules *where the following hold :*

- *ontology paths of rules in* M *occur in* $\mathcal{O}^{\mathcal{X}}$*;*
- *two rules cannot share the same label;*
- *all rules and all variables are reachable;*
- *for each pair of rules* R, R' *such that the bound variable of* R *is the root variable of* R'*, then the composition of the ontology paths of* R, R' $(op(R) \circ op(R'))$ *is a path in* $\mathcal{O}^{\mathcal{X}}$.

Definition 4.3.4 (Rule Concatenation) *Let* $R_1 : a/q_1$ *as* $v_1 \rightarrow p_1$, $R_2 : v_1/q_2$ *as* $v_2 \rightarrow p_2$ *be two rules in a mapping* M*. The result of the* concatenation *of the two rules is a new rule* $R_1.R_2 : a/q_1/q_2$ *as* $v_2 \rightarrow p_1 \circ p_2$.

For the concatenation of rules we do not define any restriction on the concatenation of the source paths of the rules. This is commensurate with the mediator-based architecture, where the mediator has no control on the local sources' schemas and is not in charge of verifying the correctness of the source descriptions (as far as the source structures are concerned). In the case of erroneous mappings, no answers will be obtained by the source[12].

Definition 4.3.5 (Closure and Expansion of Mapping) *Given a mapping* M*, its* closure*, denoted by* M^**, is the set of all rules that can be obtained from* M *by repeated concatenation. Its* expansion*, denoted* $\hat{M}$*, is the set of absolute rules in* M^* $(\hat{M} \subseteq M^*)$.

The expansion of a mapping M can be obtained by a *bottom-up fixed point computation.*

[12]Nevertheless, if the mediator is aware of the XML DTD of the source, then it is possible to verify the correctness of source paths in the mapping rules.

Interpretation of mapping rules Intuitively, mapping rules return instances of derived concepts, roles and attributes. More specifically, absolute mapping rules return instances of derived concepts and relative mapping rules return instances of concepts and role (attribute) paths.

Let r be an absolute mapping rule of the form $r: URL/q\ as\ b \rightarrow p$ where p is a *concept* c, or a *concept path*. The XML fragments obtained by evaluating the source path q of R on URL become instances of c or of the concept reached by p.

Let r be a relative rule $r: a/q\ as\ b \rightarrow p$ where p is a role/attribute path. The set of pairs of XML fragments (x, y) where x are instances of variable a, and y are instances of variable b become instances of p. Given an XML fragment x, instance of variable a, XML fragment y is obtained by evaluating XPath q on x. This rule returns also instances of the concept reached by p. These instances are the XML fragments bound in variable b.

4.4 Integrated Database

In the previous sections we have presented formally the ST_YX ontology $\mathcal{O}$ and the derived ontology $\mathcal{O}^{\mathcal{X}}$. Both databases $\mathcal{D}$ and $\mathcal{D}^{\mathcal{X}}$ of $\mathcal{O}$ and $\mathcal{O}^{\mathcal{X}}$ respectively, are *virtual* (non-materialized). The actual data resides in the sources. In this section we present how the *integrated* database $\mathcal{D}^{\mathcal{I}}$, instance of $\mathcal{O}^{\mathcal{X}}$, is obtained by the data that resides in the XML sources. Intuitively, each source, described by a set of mapping rules, defines a database instance of $\mathcal{O}^{\mathcal{X}}$ via these rules. The integrated database $\mathcal{D}^{\mathcal{I}}$ is obtained from these partial databases by first taking their union and second applying fusion as described in Section 4.2.5[13].

4.4.1 Obtaining a partial database

A source may be published by more than one mappings. From this point on we assume w.l.g. that a source is a pair (u, M) where u is the URL of the source and M is a mapping for the source. Intuitively, a source s defines a partial database denoted by $\mathcal{D}(s)$ as follows : by evaluating the mapping rules of M^*, we obtain XML fragments of the source which are represented as *objects* of $\mathcal{D}(s)$. These are labeled with *derived concepts* and are related by *derived roles*.

Let us now present more formally how the database $\mathcal{D}(s)$ is defined for a source s.

Absolute Rules : Let $R: URL/q\ as\ u \rightarrow p$ be an absolute mapping rule in M. Let X_R denote the set of XML fragments obtained by rule R by evaluating the source path q on URL. If p is a concept path, for each XML fragment in X_R, an object o is created in $\mathcal{D}(s)$ for which the following holds :

1. $\lambda_N^{\mathcal{X}}(o) = \{conc(p)\} \cup \{c \mid conc(p)\ isa^{\mathcal{X}} c\ \}$ (c is a derived concept in $\mathcal{O}^{\mathcal{X}}$ and $\lambda_N^{\mathcal{X}}(o)$ is upwards closed);
2. $\theta(o) = v$, where v is the position of the XML fragment representing o in the XML document. If the position of the XML fragment is unknown, then $\theta(o)$ is undefined.
3. $\tau(o) = v$, where v is the value of the attribute of type `ID` of the XML fragment representing o. If the fragment does not have such an attribute, then $\tau(o)$ is undefined;

Relative Rules : Let $R': u/q'\ as\ b \rightarrow\ p'$ be a relative rule in M^* such that the concatenation $R.R'$ is a rule in $\hat{M}$. $X_{R.R'}$ is the set of XML fragments obtained by evaluating the source path q on the fragments in X_R. We distinguish between two cases here :

[13]The integrated database $\mathcal{D}^{\mathcal{I}}$ is not really constructed : the global schema is not materialized and the actual data resides in the sources.

1. if p' is a role path, then for each of the fragments in $X_{R.R'}$ an object o in $\mathcal{D}(s)$ is created where $\lambda_N^{\mathcal{X}}(o) = \{conc(p \circ p')\} \cup \{c \mid conc(p \circ p')\ isa^{\mathcal{X}}\ c\}$ (c is a derived concept in $\mathcal{O}^{\mathcal{X}}$). $\theta(o)$ and $\tau(o)$ are defined as previously.
2. if p' is an attribute path, then for each XML fragment in $X_{R.R'}$ a value is added in $\mathcal{V}$. This value is obtained by casting the XML fragment to a value using one of the core functions of XPath `string()`, `number()` and `boolean()`. The function used depends on the $target^{\mathcal{X}}(p')$ in $\mathcal{O}^{\mathcal{X}}$.

Then, for all pairs of objects (o, o') where o is an object represented by an XML fragment in X_R and o' is an object/value created for some XML fragment in $X_{R.R'}$, an edge e is created where :

1. $\lambda_E^{\mathcal{X}}(e) = role(p')$;
2. $\psi(e) = (o, o')$;

These two steps are executed recursively until no new objects and edges are created. The result of this process is database $\mathcal{D}(s)$. The population of a derived concept c is the set of objects in $\mathcal{D}(s)$ labeled by c. The population of a derived role r is the set of edges in $\mathcal{D}(s)$ labeled by r. Finally, the population of a derived attribute a is the set of edges in $\mathcal{D}(s)$ labeled by a.

Example 4.4.1 *Consider source http://www.paintings.com and the set of mapping rules by which it is published, illustrated in Figure 4.6.*

Let us first consider the absolute mapping rules in the source. Set X_{A_1} contains the XML fragments obtained by evaluating the source path of the rule A_1 on URL http://www.paintings.com. *For each XML fragment in X_{A_1}, an object o is created as described above. The objects are instances of the derived concept conc(Person).*

A_1:	`http://www.paintings.com/Collection/Painter` *as* u_1	$\rightarrow$	Person
A_3:	u_1`/Painting` *as* u_3	$\rightarrow$	carried_out.produced

Consider now relative rule A_3 which can be concatenated with rule A_1. The set of XML fragments $X_{A_1.A_3}$ is obtained by evaluating the source path of A_3 on fragments in X_{A_1}. For each fragment in $X_{A_1.A_3}$, an object o is created in $\mathcal{D}(s)$ as before. The label of each object is $\lambda_N^{\mathcal{X}}(o) = \{conc(Person.carried\_out.produced), conc(Actor.carried\_out.produced), conc(Activity.produced), conc(Man\_Made\_Object)\}$. The last three derived concepts are defined by the suffixes of the path Person.carried_out.produced *and hence are superconcepts of the derived concept defined by the latter. For all pairs of objects (o, o') where o is an object created by some XML fragment in X_{A_1} and o' is an object created by some XML fragment in $X_{A_1.A_3}$, an edge e is created between o and o' whose label is $\lambda_E^{\mathcal{X}}(e) = role(carried\_out.produced)$.*

4.4.2 Obtaining the Integrated Database

In this section we present how the *integrated database* is obtained. Intuitively, the integrated database (denoted by $\mathcal{D}^I$) is obtained by first taking the disjoint union of the partial databases $\mathcal{D}(s)$ and then by applying *fusion*.

It is necessary to apply fusion for the following reasons :

1. In a source there might exist distinct objects which correspond to the same XML fragment. Consider for example source *http://www.paintings.com* and the mapping rules illustrated in Figure 4.6. Suppose that mapping rule A_9 is added in the set of mapping rules.

A_9: `http://www.paintings.com//Painting` as u_3 $\rightarrow$ Man_Made_Object

This rule states that by evaluating the source path `descendant:Painting` of the rule on the documents of source *http://www.paintings.com*, we obtain instances of concept *conc*(*Man_Made_Object*). This expression returns all XML fragments of type `Painting`, descendants of the root element of a document in the source.

Consider now the rule $A_1 \circ A_3$ produced by concatenating rules A_1, A_3.

$A_{1,3}$: `http://www.paintings.com/Collection/Painter/Painting` as u_3
$\rightarrow$ Person.carried_out.produced

Absolute rule $A_{1,3} = A_1 \circ A_3$ returns instances of the derived concept *conc*(*Person.carried_out.produced*). XML fragments that are obtained by both these rules are instances of concept *conc*(*Man_Made_Object*). Rule $A_{1,3}$ returns a subset of the XML fragments obtained by rule A_9. During the creation of the partial database, for the same XML fragment that is obtained by the two rules, two different objects are generated. To fuse the two objects there exist two choices : using the attribute of type `ID`, if the source records this information, or the position of the corresponding XML fragments in the document. If none of the above information is available, fusion using global keys must be applied.

2. Partial databases are not necessarily disjoint when these are associated with the same URL[14]. As in the previous case, two objects are created for the same XML fragment when the latter is obtained by two different mapping rules.

3. Last, in two partial databases, there might exist objects o and o' which are *globally identical*.

4.5 Query Language

Users view the mediator as a single integrated database of objects. In $STYX$ we use a *tree query language* with an OQL-like syntax to query this database as an intermediary solution that is easier to use, yet sufficiently powerful for most needs. Tree queries are based on **select-from-where** clauses on ontology paths.

Definition 4.5.1 *(**Tree Query**) Given an ontology $\mathcal{O}^{\mathcal{X}}$, a tree query Q is of the form :*

$$
\begin{array}{lll}
Q: & \textbf{select} & x_i,\ x_j,\ \ldots \\
 & \textbf{from} & p_1\ x_1, \\
 & & x_{j_2}.p_2\ x_2,\ \ldots \\
 & & x_{j_i}.p_i\ x_i,\ \ldots \\
 & \textbf{where} & c_0\ \textbf{and}\ c_1\ \textbf{and}\ \ldots
\end{array}
$$

where :

- *the x_i's are variables;*
- *each p_i in the* **from** *clause is an ontology path, called the binding path of x_i and denoted $bp(x_i)$;*
- *variable x_1 is the* root variable *of the query, and its binding path p_1 defines a derived concept in $\mathcal{O}^{\mathcal{X}}$ i.e. it is of the form c or c.r;*

[14] Recall that a source in our context is associated with a URL and a mapping M.

- *for each $i > 1$, there is a single clause $x_{j_i}.p_i\ x_i$, and p_i defines a derived role/attribute in $\mathcal{O}^{\mathcal{X}}$;*
- *x_{j_i} is called the* parent *of x_i;*
- *for all variables x_i, except the root, the composition of the binding paths of x_j and x_i where x_j is the parent of x_i, is well defined;*
- *the variables with the parenthood relation form a tree, with x_1 as its root.*

The intuition of the **from** clause is that x_1 ranges over the extent of the derived concept $conc(p_1)$, and x_i ranges over the instances obtained by traversing the derived role (attribute) $role(p_i)$ $(att(p_i))$ from the instances of x_{j_i}.

Restrictions of the query language The ST_YX query language has the following restrictions :

- The query language does not allow *joins* between query variables (in general graph queries). This restricts the expressive power of the query language but simplifies query rewriting. Nevertheless, a query that considers a join between the variables can be rewritten in two tree queries, where each one is evaluated separately against the sources, and then the join between the variables is performed at the mediator site.
- Restructuring is not allowed in the **select** clause. Restructuring may add expressive power to our language but it can be performed at the integration site. Hence it is orthogonal to the issue of retrieving data from the sources.
- The **where** clause is a conjunction of simple comparison predicates, where a simple predicate is of the form $c_i \equiv x_i \theta d$ in which $\theta \in \{=, <, >, \leq, \geq\}$ and d is an atomic value.
- The language has no quantifiers. But, a variable x_j present in the **from** clause is implicitly existentially quantified;
- The last restriction is that ontology paths appear only in the **from** clause of the query and not in the **select** or **where** clauses. It is straightforward to rewrite such a query to one where dummy variables are added in the **from** clause. For example, query Q that requests the *"names and the titles of the objects created by persons born after 1900"* can be rewritten to query Q'. Both queries are illustrated below.

Q:	**select**	x_1.carried_out.produced.has_title, x_1.has_name
	from	Person x_1
	where	x_1.born.took_place_in.year > 1900

Q':	**select**	x_2, x_3
	from	Person x_1,
		x_1.carried_out.produced.has_title x_2,
		x_1.has_name x_3,
		x_1.born.took_place_in.year x_4
	where	$x_4 > 1900$

Definition 4.5.2 *(**Query Tree**) A query Q is represented as a labeled tree $T(Q) = (X, par, bp, ops)$ where:*

- *X is the set of nodes of the tree, where each node corresponds to a variable in Q;*
- *par is the parent relation between variables (nodes);*
- *$bp(x)$ is a function returning for each variable (node) its binding path and*
- *ops is the set of operations associated with a variable x. More precisely,*
  - *for a variable x in the* **from** *clause, $\exists \in ops(x)$;*
  - *for a variable x in the* **select** *clause, $\pi \in ops(x)$,*
  - *and for each condition $x\theta d$ in the* **where** *clause, $\sigma_{x\theta d} \in ops(x)$.*

An answer to a tree query Q in some database $\mathcal{D}^{\mathcal{X}}$ can be represented as a strict type homomorphism from Q to $\mathcal{D}^{\mathcal{X}}$.

Definition 4.5.3 *(***Strict type homomorphism***) Let $\mathcal{O}^{\mathcal{X}}$ be an ontology and $\mathcal{D}^{\mathcal{X}} = (N^{\mathcal{X}}, \mathcal{V}, E^{\mathcal{X}}, \lambda_N^{\mathcal{X}}, \lambda_E^{\mathcal{X}}, \psi)$ be a database of $\mathcal{O}^{\mathcal{X}}$. Let $T = (X, par, ops, bp)$ be the query tree for a query Q. A strict type homomorphism from T to $\mathcal{D}^{\mathcal{X}}$ is a function h from nodes(variables) in T to nodes(objects) in $\mathcal{D}^{\mathcal{X}}$ such that :*

- *each node (variable) in T is mapped to some node (object) in $\mathcal{D}^{\mathcal{X}}$;*
- *if x_1 is the root variable of T then $conc(bp(x_1)) \in \lambda_N^{\mathcal{X}}(h(x_1))$;*
- *h strictly preserves the structure and the labels : for each pair of variables (x_i, x_j) where $par(x_j) = x_i$ in T then there must exist an edge $e \in E^{\mathcal{X}}$ such that $\psi(e) = (h(x_i), h(x_j))$ and $\lambda_E^{\mathcal{X}}(e) = role(bp(x_j))$ (or $\lambda_E^{\mathcal{X}}(e) = att(bp(x_j))$);*

Definition 4.5.4 *(***Valuation of a Tree Query***) A strict type homomorphism h from a query Q to a database $\mathcal{D}^{\mathcal{X}}$ is a valuation ν iff $\forall x_i \in X$ and condition $x_i\theta d \in ops(x_i)$, $h(x_i)\theta d$ is true.*

Finally, the result of a query is a set of tuples.

Definition 4.5.5 *(***Query Answer***) Let $\mathcal{O}^{\mathcal{X}}$ be an ontology and $\mathcal{D}^{\mathcal{X}}$ be a database of $\mathcal{O}^{\mathcal{X}}$. Let $T = (X, par, ops, bp)$ be a query tree on $\mathcal{O}^{\mathcal{X}}$. Let $S = \{x_1, x_2, \ldots x_n\}$ be the set of nodes (variables) where $\pi \in ops(x_i)$. Let ν be a valuation of T to $\mathcal{D}^{\mathcal{X}}$. An answer tuple t for T is defined as :*

$$t = [x_1 : \nu(x_1), x_2 : \nu(x_2), \ldots, x_n : \nu(x_n)]$$

4.6 Query Evaluation

As illustrated in Section 4.1 in order to answer a query q over a set of sources S published in the ST_YX mediator, we would like to get all answers that satisfy q. A source $s \in S$ gives *only* a *subset* of the answers for q. To get additional answers, query q is evaluated against all sources $s \in S$. When all results are obtained, *fusion* is applied at the mediator site.

To evaluate a query q on a source s, q is *rewritten* into a query that s can answer (i.e. an XML query expressed in the source's schema). In Section 4.6.1 two algorithms are presented. The result of both algorithms is a set of *variable to rule bindings* or simply *bindings* where a binding is a *vector of associations of variables to rules*. Algorithm $\mathcal{B}_f$ returns only the *full bindings* (i.e. bindings that associate *all* query variables with some rule). Algorithm $\mathcal{B}_p$ returns *maximal* bindings (i.e. bindings

that associate a subset of the query variables with some rule). A maximal binding which associates only a proper subset of the query variables with some rule, is called *partial* binding.

To complete the partial answers obtained by a partial binding, the subqueries that the source cannot answer are identified and each of them in turn is evaluated against the other sources. This process is called *query decomposition*. The *decomposition* algorithm presented in Section 4.6.2 accepts a partial binding and decomposes the query into a *prefix* and one or more *suffix* queries. The prefix query is the one for which the source provides full answers. The suffix queries are those that the source cannot answer. Section 4.6.3 presents the algorithm that calculates a *query execution plan* for query q and a set of sources S.

4.6.1 Binding variables to Rules Algorithms

In this section we illustrate two *rewriting* algorithms that accept as input a query q and a source s defined by a URL u and a mapping M, and return possibly empty sets of *variable to rule bindings*. The first, denoted by $\mathcal{B}_f$, discovers *full bindings* for the user query; the second, denoted by $\mathcal{B}_p$, discovers *maximal* bindings.

Let us first define the notion of the *prefix* of a tree.

Definition 4.6.1 (Tree Prefix) *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. 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.*

Definition 4.6.2 (Variable Binding) *A* variable to rule binding *β, or shortly* variable binding, *for a tree query Q and a source s, is a mapping from a set denoted by $dom(\beta)$ to a set denoted by $rng(\beta)$ where :*

- *$dom(\beta)$ is either empty or is the set of variables of Q or of a prefix of Q,*
- *and $rng(\beta)$ is a subset of $M^\star$ (the closure of M).*

The empty binding is denoted by β_ϕ. A binding is represented as a *vector of associations of variables to rules*, namely $[x_1 \mapsto R_1, \ldots x_n \mapsto R_n]$ where x_1 is the root of the query and each variable appears after its parent in Q.

Definition 4.6.3 (Variable Binding Properties) *The properties of a binding β for a query Q and a mapping M are the following : if $dom(\beta)$ is not empty, then β associates each variable in $dom(\beta)$ with a rule of M^*, such that the following hold:*

1. *if x is the root of query Q, then $\beta(x)$ is an absolute mapping rule such that either $conc(op(\beta(x)))$ $= conc(bp(x))$ or $conc(op(\beta(x)))$ $isa^{\mathcal{X}}$ $conc(bp(x))$, i.e., the derived concept defined by $op(\beta(x))$ is either the same or a subconcept in $\mathcal{O}^{\mathcal{X}}$ of the derived concept defined by the binding path $bp(x)$* [15],
2. *else, let $par(x) = x'$, then*
   (a) *$role(op(\beta(x)))$ $(att(op(\beta(x))))$ $= role(bp(x))$ $(att(bp(x)))$, i.e. the derived role (attribute) defined by $op(\beta(x))$, is the derived role (attribute) defined by $bp(x)$ (Condition A)* [16];

[15] Another way to formulate this condition is to say that the binding path of x ($bp(x)$) is equal to or a suffix of the ontology path of R ($op(R)$).

[16] Another way to formulate this condition is to say that the ontology path of R ($op(R)$) is the same as the binding path of x ($bp(x)$).

(b) the root variable of rule $\beta(x)$ is bound in rule $\beta(x')$ (Condition B).

Regarding the first case, if x is the root of Q, then it is bound to some derived concept by its binding path $bp(x)$, that has the form c or $c.r$. An absolute rule R can provide instances for this concept if the derived concept defined by R's ontology path ($op(R)$), is a sub-concept of the derived concept defined by $bp(x)$ in $\mathcal{O}^{\mathcal{X}}$.

The second case, states that if β is defined on x then it is defined on the parent of x (i.e. x') follows from the requirement that $dom(\beta)$ is Q or a prefix of Q. Let the declaration of x in Q be of the form $x'.l\ x$. This means that instances of variable x are obtained by evaluating the role (attribute) $role(l)$ ($att(l)$) on instances of x'. Let x' be associated in β with rule $R : u/q\ as\ v \rightarrow p$. To get instances for x, we need to find a rule R' :

1. whose ontology path defines role (attribute) $role(l)$ ($att(l)$) (condition A above);
2. and evaluates role (attribute) $role(l)$ ($att(l)$) on instances of v : the root variable of R' must be v (condition B above).

Definition 4.6.4 (Full Binding) *A binding β for a query Q and a source s is a full binding if $dom(\beta)$ is the set of variables of Q i.e. all variables in Q are associated in β with some rule in $M^\star$.*

Definition 4.6.5 (Partial Binding) *A binding β is a partial binding if $dom(\beta)$ is a proper subset of the set of variables of Q : $dom(\beta) \subset X$.*

Definition 4.6.6 (Maximal Binding) *A binding β is called* maximal *if there does not exist a binding β_1 such that $dom(\beta) \subset dom(\beta_1)$ and $\forall x \in dom(\beta)$, $\beta(x) = \beta_1(x)$.*

Computing Bindings for Acyclic Mappings We assume *acyclic mappings* and we expect M^* to be of tractable size in most practical cases, although it can be very large in some rare cases.

Two versions of the binding variables to rules algorithm are given. Both algorithms consider that the query variables are arranged in some order $x_1, \ldots, x_n$ in which the root is first, and every other node occurs after its parent, i.e., in preorder. Recall that a binding is represented as a vector of associations of variables to rules, in that order, namely $[x_1 \mapsto R_1, \ldots x_n \mapsto R_n]$. The extension of a partial binding β by $x \mapsto R$ is denoted $\beta \times [x \mapsto R]$.

Computing Full Bindings Algorithm $\mathcal{B}_f$ accepts as input the variables of a query Q arranged in preorder and the closure $M^\star$ of a mapping M. It returns a set of *full bindings*.

The algorithm is illustrated in Figure 4.15. B_i is the set of bindings calculated for up to and including x_i. In line 4 all B_i's are initialized to ϕ.

The algorithm considers first the root variable x_1 of the query and runs through the set of absolute rules in $\hat{M}$ (line 5). For each absolute rule R whose ontology path defines a derived concept, which is either the same or a subconcept of the derived concept defined by the binding path of x_1 (i.e. $conc(op(R)) = conc(bp(x_1))$ or $conc(op(R))\ isa^{\mathcal{X}}\ conc(bp(x_1))$), $[x_1 \mapsto R]$ (line 6) is added to the set of bindings for x_1 (i.e. set B_1).
The algorithm then iterates through the sequence of variables, from the left (line 8). Assume that a set of partial bindings (B_{i-1}), all defined on all variables up to and including x_{i-1}, $i > 1$, is constructed.
Consider variable x_i whose parent is variable y. Necessarily, all bindings in B_{i-1} are defined on y. This is true since the domain of a binding β is a prefix of Q. The algorithm runs through the set

1. Input: the sequence of variables of query Q, in pre-order: $x_1, \ldots, x_n$;
2. the set of mapping rules M^*;
3. Output: a set of full variable bindings;
4. Algorithm: initialization: let $B_i = \phi, i = 1, \ldots, n$.
5. loop: for each absolute rule R of $\hat{M}$ {
6. if $conc(op(R))$ $isa^{\mathcal{X}}$ $conc(bp(x_1))$ or $conc(op(R)) = conc(bp(x_1))$ then $B_1 := B_1 \cup \{[x_1 \mapsto R]\}$
7. }
8. loop: for $i = 2, \ldots, n$ {
9. loop: for each binding β_j from B_{i-1} {
10. let x_i's parent be bound by R' in β_j
11. loop : for each rule R in M^* {
12. if $role(bp(x_i))$ $(att(bp(x_i)))$ $= role(op(R))$ $(att(op(R)))$ and
13. the root variable of R is the bound variable of R',
14. then $B_i := B_i \cup \{\beta_j \times [x_i \mapsto R]\}$
15. }
16. }
17. now B_{i-1} can be discarded
18. }
19. return the set B_n

Figure 4.15: **Variable binding algorithm : $\mathcal{B}_f$**

of bindings in set B_{i-1} (line 9). For each binding β_j, assume it associates rule R' with the parent of x_i. For this binding, the set of all relative rules in $M^\star$ (line 11) are examined. Let R be such a rule. β_j is extended by $[x_i \mapsto R]$ (line 14) if :

1. $role(op(R))$ $(att(op(R)))$ $= role(bp(x_i))$ $(att(bp(x_i)))$ (line 12) (condition A, Definition 4.6.3) and
2. if the root variable of R is the bound variable of R' (line 13) (condition B, Definition 4.6.3).

Note that the edge from the parent of x_i 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 can be dropped.

Lemma 4.6.1 *Algorithm $\mathcal{B}_f$ returns all full bindings for Q.*

Proof Suppose that β is a full binding and not in B_n. Then, there exists k such that the restriction of β to the variables $\{x_1, x_2, \ldots x_{k-1}\}$ belongs in B_{k-1}. Suppose that B_{k-1} is extended for variable x_k. Suppose that the restriction of β to variables $\{x_1, x_2, \ldots x_k\}$ is not in B_k. This means that binding β does not map x_k to some rule R. This is a contradiction (β is a full binding).

Example 4.6.1 *In this example we run algorithm $\mathcal{B}_f$ for query Q_1 shown in Table 4.1, Page 100 and source* http://www.paintings.com. *The mapping rules considered are shown in Figure 4.6. Q looks for the* "title of the objects created by Van Gogh".

Query variables are arranged in preorder $\{x_1, x_2, x_3, x_4\}$ *where* x_1 *is the root variable of the query and each variable appears after its parent. We denote by* B_{x_i} *the set that contains the bindings for up to and including* x_i*, initialized in* $\emptyset$*.*

First, $\mathcal{B}_f$ *considers variable* x_1*. The absolute rule that provides answers for this variable is* A_1 *(i.e* $conc(op(A_1)) = conc(bp(x_1)) = conc(Person)$*). In this case set* B_{x_1} *is* $B_{x_1} = [x_1 \mapsto A_1]$*.*

Then, $\mathcal{B}_f$ *considers variable* x_2*, and binding* $[x_1 \mapsto A_1] \in B_{x_1}$*.* $\mathcal{B}_f$ *looks for a rule R that satisfies conditions (A) and (B) of definition 4.6.3. This rule is* A_2 *:*

1. $att(op(A_2)) = att(bp(x_2))$ *(condition A);*

2. *the root variable of* A_2 *is the bound variable of* A_1 *(i.e.* u_1*) (condition B);*

Binding $[x_1 \mapsto A_1]$ *is extended with* $[x_2 \mapsto A_2]$*. The set* B_{x_2} *is* $B_{x_2} = \{[x_1 \mapsto A_1, x_2 \mapsto A_2]\}$*. All bindings in* B_{x_1} *have been considered, and* B_{x_1} *is discarded.*

The algorithm then examines variable x_3*, and binding* $[x_1 \mapsto A_1, x_2 \mapsto A_2]$*. It finds two rules :* A_3 *and* A_8 *which satisfy conditions (A) and (B) of definition 4.6.3. Set* B_{x_3} *is* $B_{x_3} = \{[x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_3], [x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_8]\}$*. The set* B_{x_2} *is discarded.*

Last, the algorithm considers variable x_4*. The algorithm considers both bindings in* B_{x_3} *and each of the bindings is extended by rule* $[x_4 \mapsto A_4]$*. The result is set* $B_{x_4} = \{[x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_3, x_4 \mapsto A_4], [x_1 \mapsto A_1, x_2 \mapsto A_2, x_3 \mapsto A_8, x_4 \mapsto A_4]\}$*. Set* B_{x_3} *is discarded and the algorithm returns set* B_{x_4}*.*

Example 4.6.2 *Consider now query Q illustrated below. This query requests* "the title of the objects"*. Consider also source* http://www.paintings.com *and its mapping rules, illustrated in Figure 4.6.*

Q:	**select**	*b*
	from	*Man_Made_Object a,*
		a.has_title b

The algorithm starts with variable a (the root variable of the query) and looks in the expansion of the mapping rules of Figure 4.6 for a rule such that the derived concept defined by its ontology path is (i) either the same or (ii) a subconcept of the derived concept defined by the binding path of a. In the closure there is no absolute rule such that (i) holds. Nevertheless, rules $A_{1,3}$ *and* $A_{1,8}$ *illustrated below return instances of concept* Man_Made_Object *reached by some instance of concept* Person *by traversing the path* carried_out.produced*. In* $\mathcal{O}^{\mathcal{X}}$*,* $conc(Person.carried\_out.produced)$ $isa^{\mathcal{X}}$ $conc(Man\_Made\_Object)$*.*

$A_{1,3}$*:*	`http://www.paintings.com/Collection/Painter/Painting` *as* u_3
$\rightarrow$	Person.carried_out.produced
$A_{1,8}$*:*	`http://www.paintings.com/Collection/Painter/Sculpture` *as* u_3
$\rightarrow$	Person.carried_out.produced

The algorithm creates two bindings : $B_a = \{[a \mapsto A_{1,3}], [a \mapsto A_{1,8}]\}$*. After examining variable b the set of bindings returned by the algorithm is* $B_b = \{[a \mapsto A_{1,3}, b \mapsto A_4], [a \mapsto A_{1,8}, b \mapsto A_4]\}$*.*

Example 4.6.3 *Consider last query Q′ illustrated below. This query requests* "the title of the objects produced by some activity"*. Consider also source* http://www.paintings.com *published by the mapping rules illustrated in Figure 4.6 to the STYX mediator.*

Q:	**select**	*b*
	from	*Activity.produced a,*
		a.has_title b

The algorithm starts with variable a (the root variable of the query) and looks in the closure of the mapping rules of Figure 4.6, for a rule such that the derived concept defined by its ontology path is (i) either the same or (ii) a subconcept of the derived concept defined by the binding path of a. The ontology path of rules $A_{1,3}$ *and* $A_{1,8}$ *illustrated in the previous example, is* Person.carried_out.produced. *The derived concept conc(Activity.produced) defined by the binding path of a is a superconcept of conc(Person.carried_out.produced) in* $\mathcal{O}^{\mathcal{X}}$.

The algorithm creates two bindings : $B_a = \{[a \mapsto A_{1,3}], [a \mapsto A_{1,8}]\}$. *After examining variable b the set of bindings returned by the algorithm is* $B_b = \{[a \mapsto A_{1,3}, b \mapsto A_4], [a \mapsto A_{1,8}, b \mapsto A_4]\}$.

Computing Partial Bindings Algorithm $\mathcal{B}_f$, given a query Q and a source s, computes the set of *full bindings*. As illustrated by examples in Section 4.1, it might be the case that a source does not provide *full answers* for the query. We present in this section the rewriting algorithm $\mathcal{B}_p$ which calculates the set of *maximal bindings* for a user query Q and a source s. The algorithm is illustrated in more detail in Figure 4.16.

For the algorithm, query variables are arranged as for $\mathcal{B}_f$, in preorder. Similar to $\mathcal{B}_f$, a binding β is represented as a vector of associations of variables to rules. The output of the algorithm is the set B of maximal bindings for query Q and source s.

In the first step, the algorithm considers the root variable x_1 (lines 5-8) and runs through the set of absolute rules in $\hat{M}$. If R is a rule such that the derived concept defined by its ontology path ($conc(op(R))$) is a subconcept of or the same as the derived concept defined by the binding path of x_1 ($conc(bp(x_1))$) in $\mathcal{O}^{\mathcal{X}}$, then binding $[x_1 \mapsto R]$ is added in B (line 8).

$\mathcal{B}_p$ iterates through the sequence of variables (line 9) for $i > 1$. *Temp* is initialized to be the set of bindings calculated so far ($Temp = B$, line 11). Let the current variable be x_i, and y be its parent (line 12). The algorithm then iterates through the bindings in *Temp* (line 13) and examines whether variable y belongs to $dom(\beta)$. Let y be associated with R by β. $\mathcal{B}_p$ examines all relative rules in M^* (line 14). Let R be a relative rule for which conditions A and B of the Definition 4.6.3 hold. Then, binding β is extended by $[x_i \mapsto R]$ (i.e. $\beta \times [x_i \mapsto R]$) and added in set B (line 19-20). If β was extended in B with x_i, then β is removed from B (lines 21-22). When all query variables are examined, then set B is returned.

Note that the edge from y to x_i is 'traversed in this step, and only in this step'. Notice here that partial bindings are not dropped as was the case for $\mathcal{B}_f$.

Example 4.6.4 *Consider query Q illustrated below and source* http://www.paintings.com. *Q looks for* "the title of the objects created by Van Gogh along with the name and city of the museum where they are exposed, and the type and location of their images".

Q:	**select**	*d, f, g, i, j*
	from	*Person a, a.name b,*
		a.carried_out.produced c, c.has_title d,
		c.exposed_in e, e.museumName f, e.city g,
		c.image h, h.type i, h.url j
	where	*b = "Van Gogh"*

1. **Input :** the sequence of variables of query Q, in preorder: $x_1, \ldots, x_n$;
2. the closure of mapping rules M^* of some mapping M for source s;
3. **Output :** the set B of maximal bindings for Q and M
4. **Algorithm :** $B := \emptyset$;
5. for each absolute rule $R \in \hat{M}$
6. if $conc(bp(x_1)) = conc(op(R))$ or $conc(op(R))$ $isa^{\mathcal{X}}$ $conc(bp(x_1))$
7. /* $conc(op(R))$ *is a subconcept of or the same as* $conc(bp(x_1))$ */
8. add $[x_1 \mapsto R]$ to B;
9. for $i = 2, \ldots, n$ {
10. /* *Temp contains all maximal bindings up to* x_{i-1} */
11. $Temp := B$;
12. $y :=$ parent of x_i;
13. for each binding $\beta \in Temp$ where $y \in dom(\beta)$ {
14. for each rule R in M^* {
15. /* *if condition A, Definition 4.6.3 holds* */
16. if $role(op(R))$ $(att(op(R))) = role(bp(x_i))$ $(att(bp(x_i)))$ and
17. /* *if condition B, Definition 4.6.3 holds* */
18. the root variable of R is the bound variable of $\beta(y)$
19. /* β *is extended to* x_i *and added in* B */
20. add $\beta \times [x_i \mapsto R]$ to B;
21. if β was extended to x_i
22. remove β from B;
23. }
24. }
25. return B;

Figure 4.16: **Variable binding algorithm** $\mathcal{B}_p(Q, s)$

For the subquery that requests "the title of the objects created by Van Gogh" *the source returns two full bindings as illustrated in Example 4.6.1 :* $B = \{[a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4], [a \mapsto A_1, b \mapsto A_2, c \mapsto A_8, d \mapsto A_4]\}$.

$\mathcal{B}_p$ *considers variable* e *and binding* $\beta_1 = [a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4]$. *The parent variable of* e *(i.e.* c*) belongs to* $dom(\beta_1)$. *The algorithm runs through all the relative rules in* M^* *to look for a rule* R *such that :*

1. *the ontology path of* R *is* exposed_in *(i.e. the binding path of* e*) and*

2. *its root variable is the bound variable of* A_3 *(notice that* $dom(\beta_1)$ *binds* c *to* A_3*).*

There is no such rule in M^*. *Hence, binding* β_1 *is left in* B. *For binding* $\beta_2 = [a \mapsto A_1, b \mapsto A_2, c \mapsto A_8, d \mapsto A_4]$, *the same holds, so binding* β_2 *is left in* B.

Then variables f *and* g *are considered. Their parent (i.e.* e*) does not belong to the set of variables mapped by any of the two bindings in* B *and hence,* B *does not change.*

The algorithm considers variable h *and runs through the bindings in* $Temp = B$. *It considers first binding* $\beta_1 = [a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4]$. *Variable* c *is mapped to* A_3. *The relative rule of* M^* *whose (i) ontology path is* image *(i.e. the binding path of* h*) and (ii) root variable is the bound variable of* A_3 *is rule* A_5. *The same holds for binding* $\beta_2 = [a \mapsto A_1, b \mapsto A_2, c \mapsto A_8, d \mapsto A_4]$. *In this case,* $B = \{[a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4, h \mapsto A_5], [a \mapsto A_1, b \mapsto A_2, c \mapsto A_8, d \mapsto A_4, h \mapsto A_5]\}$. *Notice that the partial bindings which have been extended are removed from* B.

For variable i, *there is no rule such that its ontology path is* type *(i.e. the binding path of* i *in the query). In this case,* B *does not change.*

Last, the algorithm considers variable j. *Each of the bindings in* $Temp = B$ *is extended by* $[j \mapsto A_6]$. *Rule* A_6 *is a relative rule whose ontology path is* url *(i.e. the binding path of* j *in the query). The set of bindings returned by the algorithm is* $B = \{[a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4, h \mapsto A_5, j \mapsto A_6], [a \mapsto A_1, b \mapsto A_2, c \mapsto A_8, d \mapsto A_4, h \mapsto A_5, j \mapsto A_6]\}$. *Again, the partial bindings which have been extended have been removed from* B.

Optimization Both algorithms make an extended use of the closure of a mapping M. They examine all relative rules of $M^\star$ in order to discover those that can provide answers for the query variables. 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 $role(bp(x_i)) = role(op(R))$. Thus, all we need to do for this case is to consider all relative rules of M^* for which the length of the ontology path is limited by $\mid bp(x_i) \mid$. Recall that each rule in M^* is a concatenable sequence of rules of M, and that relative rules of M^* have ontology paths longer than zero. It follows that in the algorithm above, it suffices to use only the relative rules of M^* whose ontology path length is limited by $\mid bp(x_i) \mid$, where $\mid bp(x_i) \mid$ is the maximum length of $bp(x_i)$.

4.6.2 Query Decomposition

Algorithm $\mathcal{B}_p$ presented previously returns a set of *maximal bindings*. In the case of a *partial* binding we do not obtain *full answers* from the source.

Intuitively, if β is a partial binding and $\beta \in \mathcal{B}_p(Q, s)$, and x is a variable that belongs to $dom(\beta)$ and has a child y in Q which does not belong to $dom(\beta)$, then for an instance of x an instance for y is missing. To find the missing answers for x we consider for evaluation the query that for an instance of x looks for instances of y. When the two queries are evaluated the results are then *joined*.

More generally, for a partial binding β, query Q is *decomposed* into a *prefix query* (which corresponds to β) and to one or more *suffix* queries.

Definition 4.6.7 (Query Decomposition) *Let Q be a query, and s a source. Let partial binding $\beta \in \mathcal{B}_p(Q, s)$. The operator decompose, denoted by δ, accepts as inputs β and Q and returns a prefix query $Q_p(\beta)$ and a set of suffix queries $\mathcal{QS}(\beta)$: $[Q_p(\beta), \mathcal{QS}(\beta)] = \delta(Q, \beta)$ where :*

- *$Q_p(\beta)$ is a prefix of Q such that variables of $Q_p(\beta)$ are variables in $dom(\beta)$;*
- *Let x be a variable that belongs to $dom(\beta)$, and let $S = \{y_1, y_2, \ldots y_n\}$ be its children in Q that are not in $dom(\beta)$. Then a suffix query $Q_s(\beta)$ with root x is defined as follows :*
  1. *$Q_s(\beta)$ contains the set of variables $\{x, y_1, y_2, \ldots y_n\}$, and all descendants of each y_i in Q;*
  2. *the set of edges of $Q_s(\beta)$ is the set of edges of Q restricted to that set of variables;*
  3. *the binding path of x in $Q_s(\beta)$ is the target concept of its binding path in Q.*

Example 4.6.5 *Consider binding β, result of the rewriting algorithm $\mathcal{B}_p$ in Example 4.6.4 :*

$$\beta = [a \mapsto A_1, b \mapsto A_2, c \mapsto A_3, d \mapsto A_4, h \mapsto A_5, j \mapsto A_6]$$

Given this binding and the initial query, variables e and j do not belong to β, but their parents (c and h respectively) are in $dom(\beta)$. The prefix query $Q_p(\beta)$ and the two suffix queries $Q_{s_1}(\beta)$ and $Q_{s_2}(\beta)$ are illustrated below.

$Q_p(\beta)$:	**select**	*d, j*
	from	*Person a, a.*has_name *b,*
		*a.*carried_out.has_produced *c, c.*has_title *d*
,		*c.*image *h, h.*url *j*
	where	*b = 'Van Gogh'*

$Q_{s_1}(\beta)$:	**select**	*f, g*	$Q_{s_2}(\beta)$:	**select**	*i*
	from	Man_Made_Object *c,*		**from**	Image *h,*
		*c.*located_at *e,*			*h.*type *i*
		*e.*museumName *f,*			
		*e.*city *g*			

The root of query $Q_{s_1}(\beta)$ is variable c, and its binding path is Man_Made_Object. *Concept* Man_Made_Object *is the target of the binding path of c (*carried_out.produced*) in query Q. Similarly, for query $Q_{s_2}(\beta)$, its root is variable h whose binding path is concept* Image, *the target of the binding path of h (*image*) in Q.*

Extending Queries with Global Keys Recall from the examples in Section 4.1.4 that in order to complete the partial answers obtained from the prefix query, these must be joined with the answers obtained from the suffix queries. To perform these joins, we need to extend all queries (prefix and suffix) with keys.

Let $Q_p(\beta)$ be the prefix query and let Q_i be a suffix query in $\mathcal{QS}(\beta)$. Let x be the root variable of Q_i and the binding path of x in Q_i be concept c in $\mathcal{O}^{\mathcal{X}}$. Then, for a global key k of c the prefix and the suffix queries are extended as follows :

- for all attribute paths a_i in k, add in the queries **from** clause, the statement $x.a_i\ t^c_{a_i}$ where $t^c_{a_i}$ is a new distinct variable which binds the values obtained by following attribute path a_i from instances of x;
- add in the query **select** clause variables $t^c_{a_i}$ defined previously.

Remarks We have to note here that if the root variable of the query is bound to a concept which has more than one keys, for each such key a new 'extended' prefix and suffix query must be created. When the results are obtained for each such pair of queries, a join and then a union must be performed. From this point we assume that each concept is associated with only one global key.

Example 4.6.6 *Consider the prefix and the suffix queries in Example 4.6.5. The root variable of the suffix query $Q_{s_1}(\beta)$ is variable c bound in concept* Man_Made_Object. *The key for concept* Man_Made_Object *is K(Man_Made_Object) = $\{\{museumIdentifier\}\}$. In this case for key $k = \{museumIdentifier\}$, the prefix and the suffix query $Q_{s_1}(\beta)$ extended with k, are shown below.*

$Q_p(\beta)$:	**select**	d, j, t_1
	from	*Person* a, a.has_name b,
		a.carried_out.has_produced c, c.has_title d
,		c.image h, h.url j,
		c.***museumIdentifier*** t_1
	where	b = *'Van Gogh'*

$Q_{s_1}(\beta)$:	**select**	f, g, t_1
	from	Man_Made_Object c,
		c.located_at e,
		e.museumName f,
		c.***museumIdentifier*** t_1
		e.city g

4.6.3 Generation of Query Execution Plans

This section shows how to generate a *query execution plan* for a query Q and for a set of sources S. Intuitively, given a query Q and a set of sources S, Q is evaluated against each source in S. When the results are obtained from all the sources, they are *unioned* at the mediator. During this union, duplicates i.e. objects which are either locally or globally identical, are not eliminated. On this collection of objects, the fusion operator, defined in Section 4.2.5, Page 119, is applied.

We have shown previously that when a query Q is rewritten for a source s, it might be the case that a *partial binding* is obtained. In this case, the query is decomposed into a *prefix* query and one or more *suffix queries*. Each one of the latter is considered again for rewriting, for each of the sources in S. Before showing how a query execution plan is defined for a query Q and a set of sources S, let us define what is a *rewriting* for a query Q and a source s.

Let Q be a query and s a source. Let β be a partial binding produced when rewriting Q for s using algorithm $\mathcal{B}_p$ ($\beta \in \mathcal{B}_p(Q, s)$). Let $\delta(Q, \beta) = [Q_p(\beta), \mathcal{QS}(\beta)]$ be the result of the decomposition of Q for β where $Q_p(\beta)$ is the prefix query and $\mathcal{QS}(\beta)$ is the set of suffix queries. Query $Q_p(\beta)$ can be translated into an XML query using binding β. For each suffix query in $\mathcal{QS}(\beta)$, when this is considered for rewriting, either a full binding is found, or the suffix query has still to be decomposed.

We call a *β-query rewriting* $\mathcal{LR}(Q, \beta)$, for a decomposition $\delta(Q, \beta)$, the *join* between the prefix query $Q_p(\beta)$ and all suffix queries $Q_i \in \mathcal{QS}(\beta)$. It is evident that if β is a full binding $\mathcal{LR}(Q, \beta) = Q$.

Definition 4.6.8 (β-query rewriting) *Let $\delta(Q,\beta) = [Q_p(\beta), \mathcal{QS}(\beta)]$ be a decomposition for Q and binding β. The β-query rewriting denoted by $\mathcal{LR}(Q,\beta)$ is defined as :*

$$\mathcal{LR}(Q,\beta) = Q_p(\beta) \bowtie_{LG} \mathcal{GR}(Q_1,S) \bowtie_{LG} \ldots \bowtie_{LG} \mathcal{GR}(Q_n,S)$$

where $\mathcal{GR}(Q_i,S)$ is an **S-query rewriting**.

Definition 4.6.9 (S-query rewriting) *Let Q be a query and S a set of sources. An S-query rewriting for Q and S, denoted by $\mathcal{GR}(Q_i,S)$, is the union of the* **β-query rewritings** *for all sources in S :*

$$\mathcal{GR}(Q,S) = \bigcup_{s \in S} \bigcup_{\beta \in \mathcal{B}_p(Q,s)} \mathcal{LR}(Q,\beta)$$

A *query execution plan* QEP is defined as follows : (1) a query q that can be answered by a single source (that is a query for which there exists a full binding) is a(n atomic) QEP; (2) the union of two QEP's is a QEP (3) the join of two QEP's is a QEP[17]. Basically, sources answer atomic queries in a QEP and the mediator performs joins and unions. A QEP can involve several atomic queries sent to a given source. It might be interesting to combine such queries in a single query. This implies the reorganization using classical properties such as distributivity of union w.r.t. join.

1. **Input:** a query Q and a set of sources S
2. **Output:** a query execution plan for Q;
3. **Algorithm:** $GQEP(Q,S) = \emptyset$;
4. for all sources $s \in S$ {
5. if $\mathcal{B}_p(Q,s) \neq \emptyset$ {
6. /* *there exists at least one maximal binding for Q in s* */
7. for all bindings $\beta \in \mathcal{B}_p(Q,s)$ {
8. if β is a full binding $P(\beta) := Q$;
9. else { $P(\beta) := Q_p(\beta)$;
10. for all suffix queries $Q' \in \mathcal{QS}(\beta)$
11. if $P(\beta) \neq \emptyset$
12. /* *there exists a non-empty query plan* */
13. /* *for all subqueries up to Q'* */
14. if $GQEP(Q',S) \neq \emptyset$
15. /* *there exists a query plan of Q'* */
16. $P(\beta) := P(\beta) \bowtie_{LG} GQEP(Q',S)$;
17. else $P(\beta) := \emptyset$;
18. }
19. $GQEP(Q,S) = GQEP(Q,S) \cup P(\beta)$
20. }
21. }
22. }
23. return $GQEP(Q,S)$;

Figure 4.17: Query Execution Plans Generation

[17] Join is non commutative : the root variable of the second QEP should be one of the variables in the first QEP.

Given a set of sources S and a query Q, the algorithm P(Q) shown in Fig. 4.17 computes a query execution plan for Q. The algorithm runs through the set of sources in S. For each source s and binding $\beta \in \mathcal{B}_p(Q, s)$, a QEP $P(\beta)$ of the prefix rewriting $\mathcal{LR}(Q, \beta)$ is computed: if β is a full binding (i.e. complete answers are obtained), the result is query Q (line 8). Else, if β is a partial binding, then Q is decomposed into a prefix query $Q_p(\beta)$ and a set of suffix queries $\mathcal{QS}(\beta)$ (these queries are also extended by adding appropriately the keys as shown in Section 4.6.2). The query execution plan of Q against source s is obtained by joining $Q_p(\beta)$ with the query execution plan for each suffix query $Q' \in \mathcal{QS}(\beta)$ (line 16). To calculate the query execution plan of a suffix query Q' the algorithm is called recursively. Finally the obtained plan is added to the existing plan by union (line 19).

It is easy to see that a source s cannot provide answers for a query Q in the following cases :

- if the binding variables to rules algorithm $\mathcal{B}_p(Q, s)$ returns an empty set of bindings;
- if there exist some suffix query $Q_i \in \mathcal{QS}(\beta)$ that cannot be answered by the set of sources in S. This knowledge can be used for pruning the recursive evaluation of the above algorithm.

Another issue that can be considered here for pruning the recursive evaluation of the suffix queries in the query execution plan is the presence of keys. For example, consider a source s and a query Q. Let $Q_p(\beta)$ be the prefix query resulting from applying the decomposition $\delta(Q, \beta)$. To be able to complete the missing answers for the prefix query we must add the keys on the variables on which we will perform the joins. If source s returns values neither for the local keys nor for the global keys, then there is no point in considering the recursive evaluation of the suffix queries.

4.7 Comparing ST_YX and Xyleme

In this section we discuss the choices made for the Xyleme [55] and ST_YX systems. The purpose of the Xyleme system is to build a datawarehouse for the XML data in the Web. The XML documents are crawled from the Web and stored in the Xyleme repository. One of the objectives of the system, presented in detail in Section 2.4.2, was to provide a *single access* to all the XML documents in the Xyleme repository, by hiding their heterogeneities to the end-user. XML documents are described by DTDs called *concrete DTDs*. As in standard data integration systems, the objective here is to relieve the user from querying each concrete DTD to obtain the answers to her queries. The user queries the XML documents stored in the repository by formulating simple tree queries defined in terms of a set of *abstract DTDs*. These DTDs, which are simple tree structures, constitute a *view* on top of the concrete XML DTDs.

In ST_YX the objective was to support the *querying* and *integration* of heterogeneous and autonomous Web XML resources. In contrast to Xyleme where XML resources are crawled and stored in Xyleme's repository, in ST_YX XML resources remain accessible on the Web. Users formulate queries in terms of the ST_YX global schema which is an *ontology*. The ontology is a *symmetric* schema, where *concepts* are related with *roles* and *inheritance* relationships, and are associated with *attributes*.

In the discussion that follows, Xyleme's abstract DTDs and the ST_YX ontology are refereed to as *global schema*. A common point in Xyleme and ST_YX is that a source is described to the global schema by means of a set of mapping rules which map *source paths* to *global schema paths*. In contrast, in a number of data integration systems following the local as view approach [135] a source is described to the mediator by a *description* which is a *conjunctive* query defined in terms of the global schema relations.

In this section two issues are addressed: (i) the choice of the *global schema* and (ii) the choice of the *mapping language* in $STYX$ and Xyleme.

4.7.1 Global Schema

A number of data integration projects [136, 81, 175, 149] and the relevant theory were presented in the framework of the relational model. When the sources are relational, using the relational model for integration has some advantages; in particular the same language can be used as a query language as well as the language to define the *source to global schema* mappings. The advantages of the relational model are well known and in particular relational query languages can be used for the (i) the source descriptions, (ii) user queries and (iii) query rewritings. We have to note that even in this framework, it is often desirable to use additional mechanisms to specify constraints on mappings, or to describe source capabilities. However, we are concerned with sources that are XML and not relational. The decision which model to use for the $STYX$ global schema data model in such a context is not easy to make. An obvious question is "Why not XML?".

Indeed Xyleme [55] uses XML for the global schema data model. *Abstract DTDs* provide a hierarchical view on a collection of heterogeneous concrete DTDs and are *simple tree structures*. There is no notion of element or attribute, and the horizontal relationships found in XML DTDs, which are modeled using the XML `IDREF` attribute mechanism, are not permitted. We discuss here the shortcomings of using an XML DTD as a global schema[18] in general.

A $STYX$ ontology distinguishes between *concepts*, *roles* and *attributes*. Concepts represent real world entities (objects), related to each other by *symmetric roles*. Concepts are also associated with *values* by *attributes*. Finally, concepts are related by the *isa* relationship which describes commonality of structures and has subset semantics. Finally, instances of a concept can be identified by using *global* or *semantic keys* defined at the ontology level and *local* keys, defined at the XML document level.

The basic constructs in XML DTDs are *elements* and *attributes*. Element nesting is a *hierarchical* parent/child relationship. XML DTDs also record *horizontal relationships* using the `ID/IDREF` attribute mechanism. But, these relationships are *untyped* : there is no way to specify that an attribute of type `IDREF` is of type a, where a is an XML element. Finally, XML DTDs do not support *isa* relationships.

The important issues in our context are the *symmetric* and *isa* relationships. Let us discuss each in turn.

With XML, a symmetric binary relationship is modeled by an asymmetric parent-child relationship between two elements. A source can choose either of the two element types as being the parent of the other. For example, `Painting` is a child of `Painter` in the source *http://www.paintings.com*. Another source might choose to invert this relationship, so `Painting` becomes a parent of `Painter`. Of course, binary relationships may as well be represented by the XML `ID/IDREF` attribute mechanism. But again these are asymmetric and there is no way to specify that one relationship is the inverse of another. If XML was chosen as the global schema data model, and element nesting as the representation of relationships, this problem of inverse hierarchies between a global schema and a source would force the use of the ancestor axis of XPath in the global schema side. This would not only render the source description complex but also significantly complicate query processing. The presence of symmetric relationships, where each one has an inverse, and either direction can be used, simplifies the formulation of queries and mapping rules.

[18] Although in the XML Schema proposal [198], some shortcomings of XML DTDs discussed here are partially solved, we believe that the following arguments merit attention.

isa hierarchies are another feature of our ontology, useful for data modeling of the domain as well as for integration scenarios. For data modeling, the *isa* hierarchies represent commonality of structures, and help in the resolution of the terminological differences between sources. For example, the Information Manifold project [136] uses *class hierarchies* from Description Logic languages to describe differences between the contents of information sources. To illustrate its querying power, it allows users to request information about persons, or about special kinds of persons, such as painters. This would not be possible had we used only one kind, say person, in the global schema, and mapped all kinds of persons from sources to it. Note also that when a user query requests information about persons, information about painters will be returned, since the set of painters is a subset of the set of persons. On the other hand, if a user requests information about painters, information about sculptors will not be returned.

4.7.2 Mapping Language

As mentioned earlier, the principle of the mapping language for *STYX* and Xyleme is the same : an XML resource is described to the global schema by means of mapping rules which associate *source paths* to *global schema paths*.

Nevertheless, the problematic of Xyleme and ours are different. As presented in Section 2.3.2, the purpose of the Xyleme system is to build a Web scale data warehouse where all data is materialized. In this case, the capabilities of sources with respect to the query language supported is not a concern of the system. The Xyleme system is in charge for the *complete* evaluation of the user query.

In contrast to Xyleme, in our context the actual data resides in the sources. A user query must be rewritten into one or more XML queries, which are sent for evaluation to the XML resources. When this work started, there existed a limited number of XML resources available on the Web. These resources were basically XML documents stored in some Web server and the only way to access them was by their URL. Hence, these sources were not able to evaluate any XML queries. Nevertheless, software such as Fragserver[19], a Java servlet installed very easily on top of Web servers are able to evaluate simple XPath expressions. Moreover, XPath is already part of other XML-related languages [4] for the *transformation* (XSLT [82]), *linkage* (XLink [69]) and *querying* (XQL [184], XQuery [44] and Quilt [45]) of XML documents.

Concerning the mapping language, in Xyleme source (*concrete*) and global schema (*abstract*) paths are absolute paths (i.e. specified only from the root of the abstract/concrete DTD) and use *only* the *child* axis. In *STYX* source paths are *XPath location paths* which are evaluated on a variable or on a URL.

The difference between *STYX* source paths and Xyleme concrete paths is that for the former the XPath language is used. Hence, one can specify source paths using (i) any of the XPath axis (i.e. parent, ancestor, descendant, sibling, attribute etc.), and (ii) functions from the core library of XPath (i.e. `id()`). One can navigate in any direction in the XML document. Moreover, more complex expressions considering the *order* of the XML document can also be specified. For example, if one knows that the first element x in a document is an instance of concept c and the second an instance of concept c' then one can write the following rules :

R:	`URL`/x[1] as u_1	$\rightarrow$	`c`
R':	u_1/`following-sibling::x` as u_2	$\rightarrow$	`c'`

The use of the function `id()` of XPath allows one to navigate using the horizontal relationships specified using the attributes of type `IDREF`. Consequently, the use of XPath in the *STYX* mapping

[19] `http://www.xml.com/pub/r/676`.

rules allow one to describe *any kind* of XML structure. This is not the case in Xyleme : for example one cannot express concrete paths which consider the order of elements in XML documents. Moreover, XML DTDs which use the `ID/IDREF` attribute mechanism cannot be described.

Finally, consider global schema paths. In Xyleme abstract paths, similar to concrete paths, are defined from the *root* of the abstract DTD and use only the child axis. In $STYX$ the ontology path language is much more expressive. Since the $STYX$ ontology is a graph, one can enter at *any* point of the graph and navigate in any direction using the roles defined in concepts. The presence of a symmetric schema supports a more natural formulation of the mapping rules and queries.

4.8 The $STYX$ Prototype

In this section we present the architecture of the $STYX$ prototype implemented to demonstrate our approach for the querying and integration of XML Web resources. A *three level* architecture was used for our prototype. The user is able to *navigate* in the schema in order to *formulate* her queries. The client sends specific requests to a *Java servlet* which runs on a Cocoon-enabled Apache Server. This servlet is responsible for sending the requests (navigation or queries) to the $STYX$ mediator which evaluates the query. The latter returns to the servlet the answers in the form of XML documents. The $STYX$ mediator is responsible for the *loading* of the global schema. It is responsible for the *analysis* and *parsing* of the query, the generation of the *query execution plans* and their *evaluation* against the published XML resources. When the final results are obtained, they are sent to the Cocoon servlet which processes and transforms the documents to HTML pages which are then presented to the navigator of the user. The communication between the navigator of the client and the servlet is done by HTTP and the servlet communicates with the $STYX$ mediator using RMI (remote method invocation) Java objects.

The choice of this three layer architecture offers a certain flexibility since we are not bound by the functionalities and implementations of the information providers. The programming language is Java. We followed a main memory implementation where (i) global schema contents and mapping rules are stored as Java objects, and (ii) query processing is done in main memory. Java objects are communicated between the different modules of the $STYX$ mediator. One of our objectives in the development of $STYX$ was to follow and exploit as much as possible standard XML technologies and recommendations such as XQuery, XPath and XSLT. It shows once more that XML is not only a flexible format for data exchange but has become a mature technology for building Web portals.

4.8.1 $STYX$ System Architecture

The architecture of the system is presented in Figure 4.18. The various functionalities implemented by this architecture are the following:

XML Resource Publishing : XML resources are published in the $STYX$ mediator by mapping *manually* XML source fragments (specified by *XPath location paths*) to *paths* of the $STYX$ *ontology* using the **Source Publication Interface** which interacts with the $STYX$ ontology (*portal schema*) through the **Schema Manager.** The functionalities offered by the Source Publication Interface are (i) the *modification* of *existing* mappings and (ii) the on-line *creation* of new mappings for an existing source and (iii) the *publication* of a new source. In the two first cases the user is able to *insert* new mapping rules, and *modify* or *delete* existing ones. The inserted or modified rules are sent to the **Rules Manager** which is responsible for the validation and the storage of the mapping. When a XML resource is published it also provides a XML Stylesheet (XSLT) [82] that specifies how source data can be displayed.

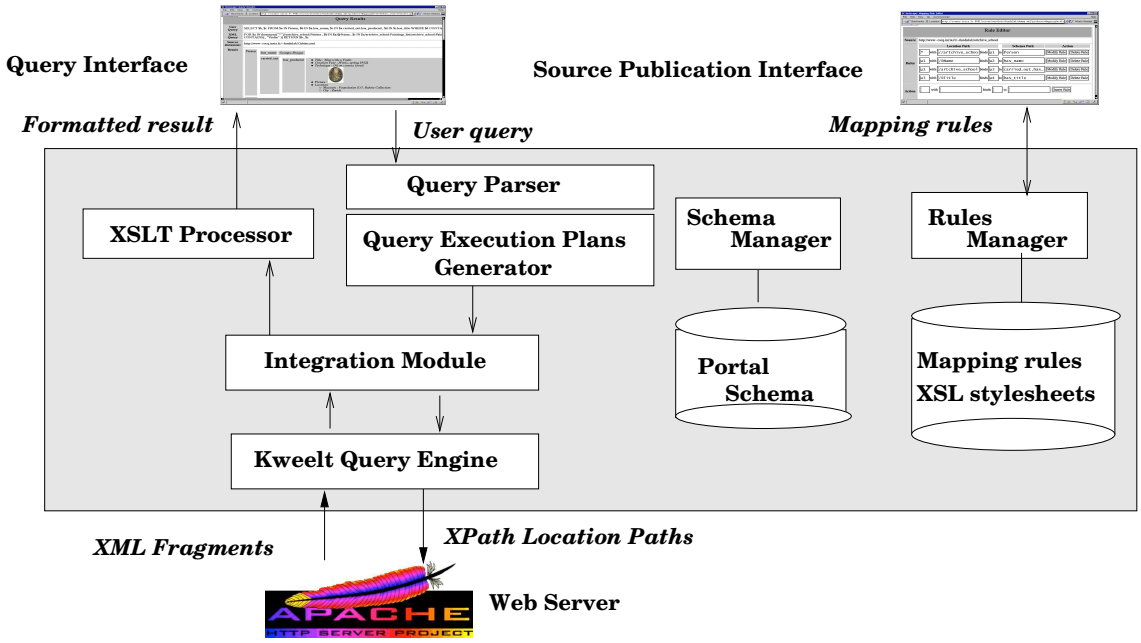

Figure 4.18: $STyX$ System Architecture

The storage and validation of the mappings is done by the **Rules Manager**. This module is responsible to verify whether the mappings satisfy the conditions specified in Definition 4.3.3 (Page 121).

Query Processing : is done in several steps. Figure 4.19 illustrates in detail the query processing steps in the $STyX$ mediator :

1. *User queries* are formulated using the **Query Interface** or are simply stored in form of a hypertext link (URL). The user formulates her queries in an OQL-like syntax based on **select-from-where** clauses. The **Query Interface** is a Web service that forwards the query to **Query Parser**. The **Query Parser** accepts the query and :
   (a) first, performs a syntactical analysis of the query;
   (b) second, verifies if it is *well typed* : (i) if it is a *tree query*, (ii) if the binding paths of the variables are paths in the ontology and (iii) for all pairs of variables (x, y) where x is parent of y, the composition of their binding paths is a path in the ontology.
   (c) third, produces a language neutral intermediate representation of the user query, which is then forwarded to the **Query Execution Plans Generator**.
2. The *generation of query plans* is done by the **Query Execution Plans Generator** module. It accepts the internal representation of the user query and constructs a *query execution plan* using the algorithm $P(Q)$. The resulting plan is a *union* of the *plans* for the query and each of the sources published in the $STyX$ mediator. Each plan for the query and for a source considers either *joins* between the *prefix query* and the *suffix* queries resulting from the decomposition of the query given a *partial binding* or simply a *prefix* query given a *full binding*. To calculate the execution plan, the **Query Execution Plans Generator** module interacts with the **Rules Manager** and the **Schema Manager**.

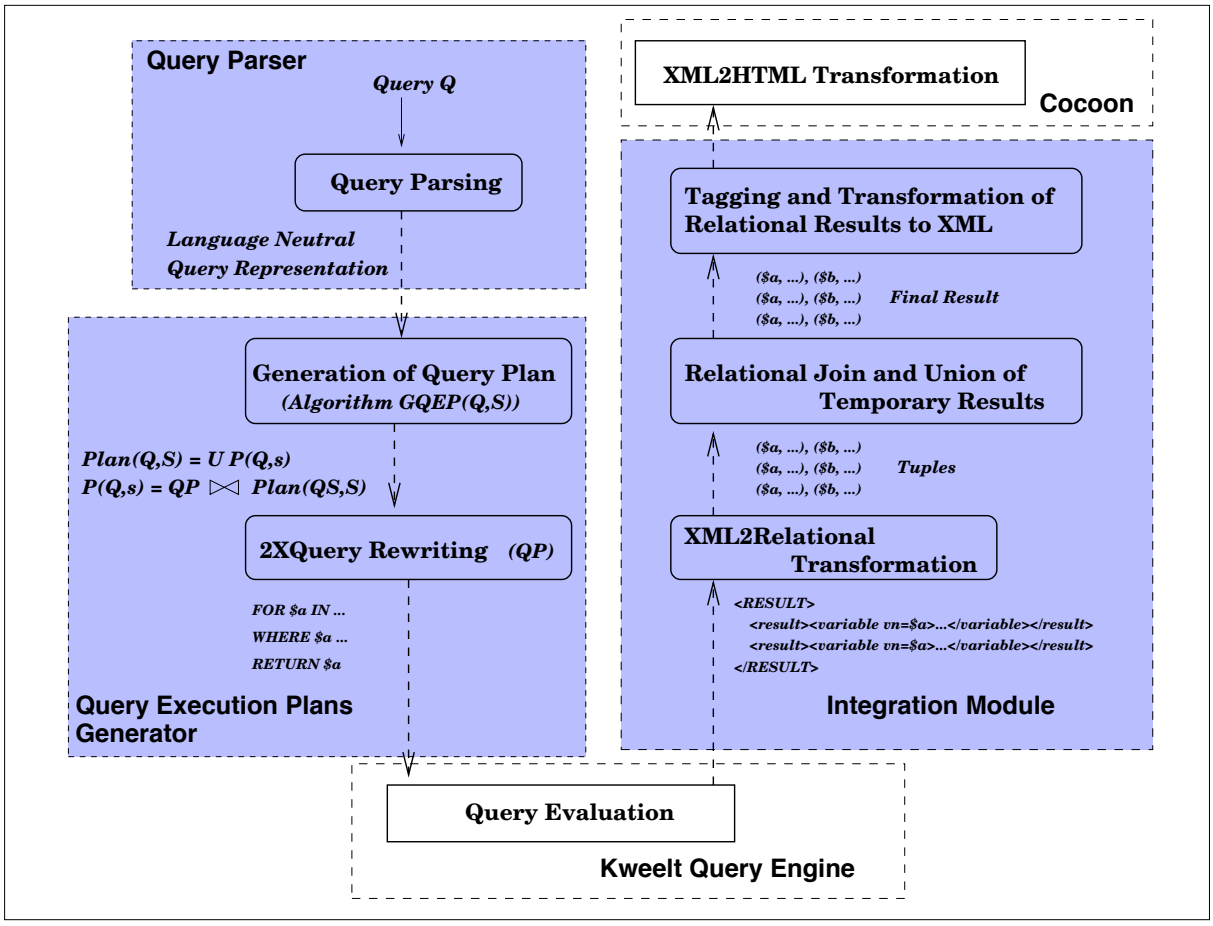

Figure 4.19: Query Processing in $STYX$

The **Query Execution Plans Generator** runs through this expression, and translates each query using the calculated binding, to an XQuery expression. This is sent to the **Kweelt Query Engine** which is responsible for the evaluation of the query on the source. The result of the evaluation is an XML document. This is forwarded to the **Integration Module** which is responsible for the transformation of this *intermediate* result to a set of tuples and its storage.

When all the queries in a plan for the initial user query and a source are evaluated the integration module performs the necessary *joins*. When the query is evaluated against all sources, then the temporary results are *unioned* by the **Integration Module.**

This module uses the initial query, and transforms the set of tuples to an XML document.

Result Display : The results obtained by the source queries are reformatted before being returned to the user : first the **XML tagger** inserts schema specific tags using the mapping rules and the **XSLT processor** finally transforms the result into an HTML document (or any other format defined by the XSLT stylesheet) which can be displayed to the browser of the user.

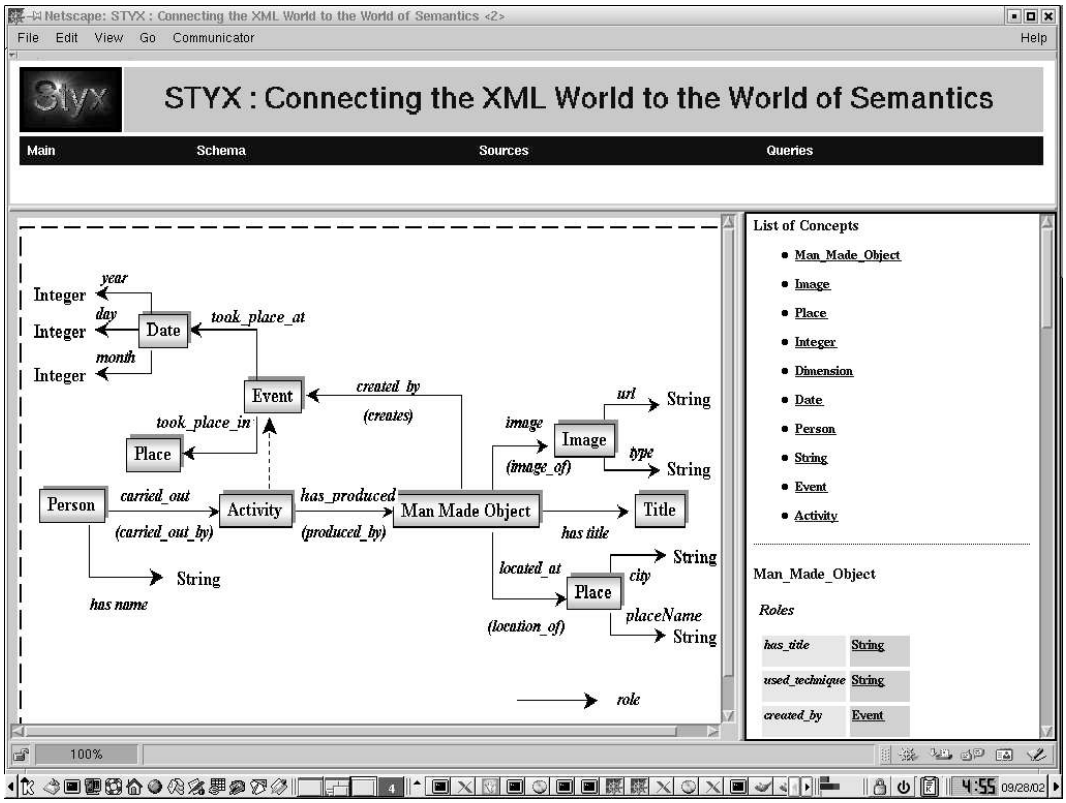

Figure 4.20: *STYX* Global Schema

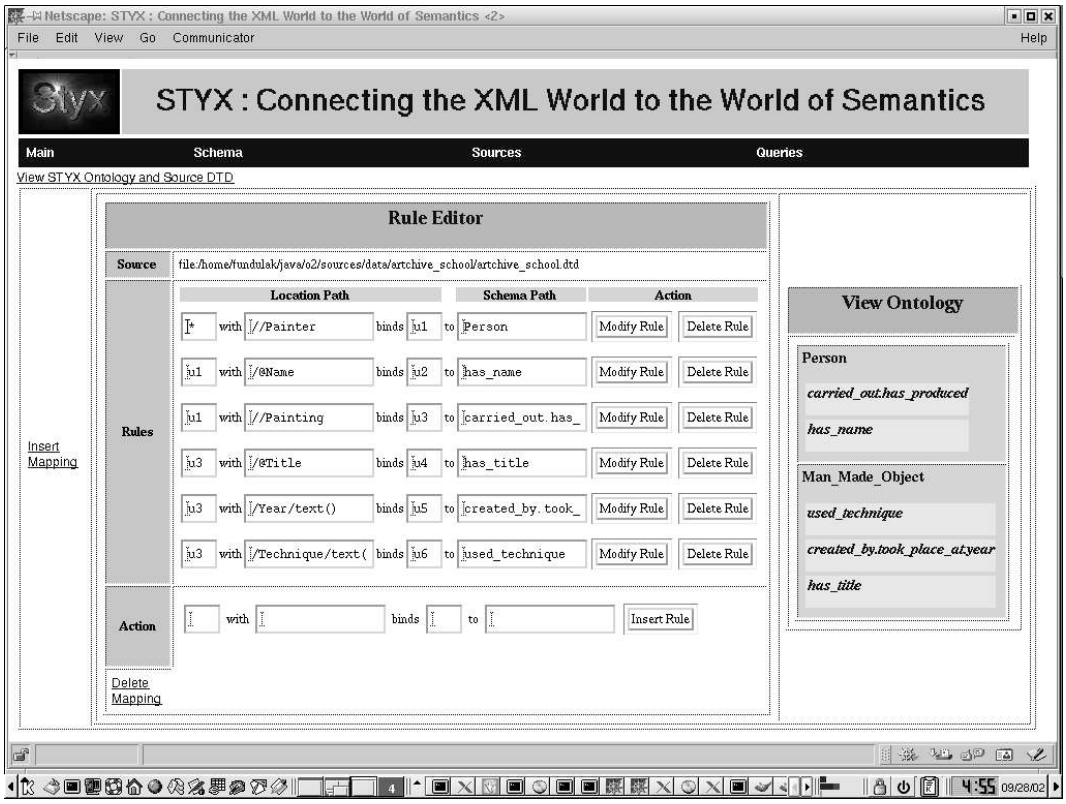

Figure 4.21: *STYX* Mapping Rules

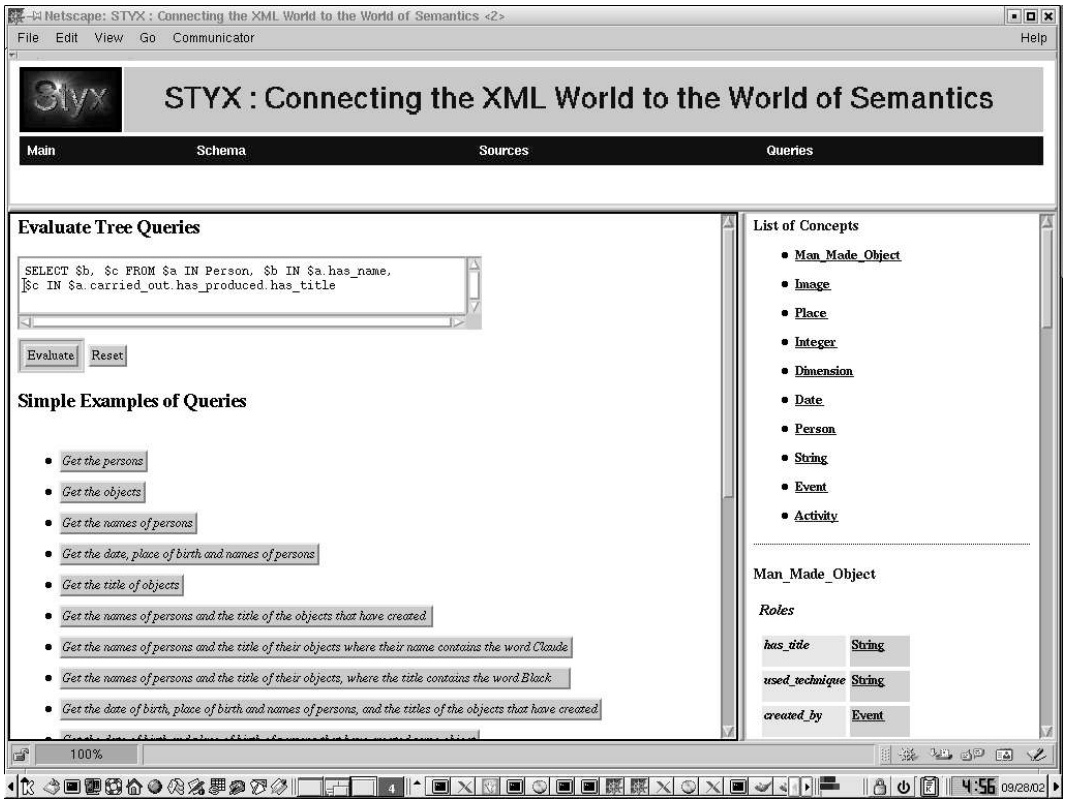

Figure 4.22: *STYX* Query Interface

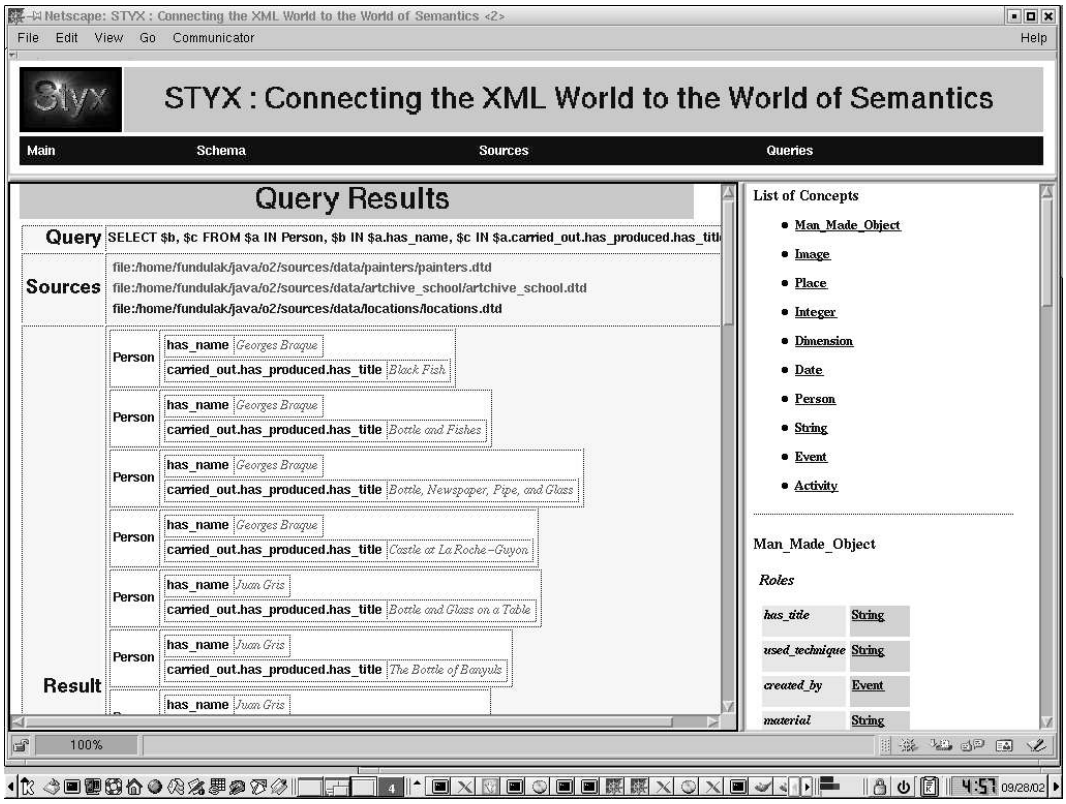

Figure 4.23: Query Results

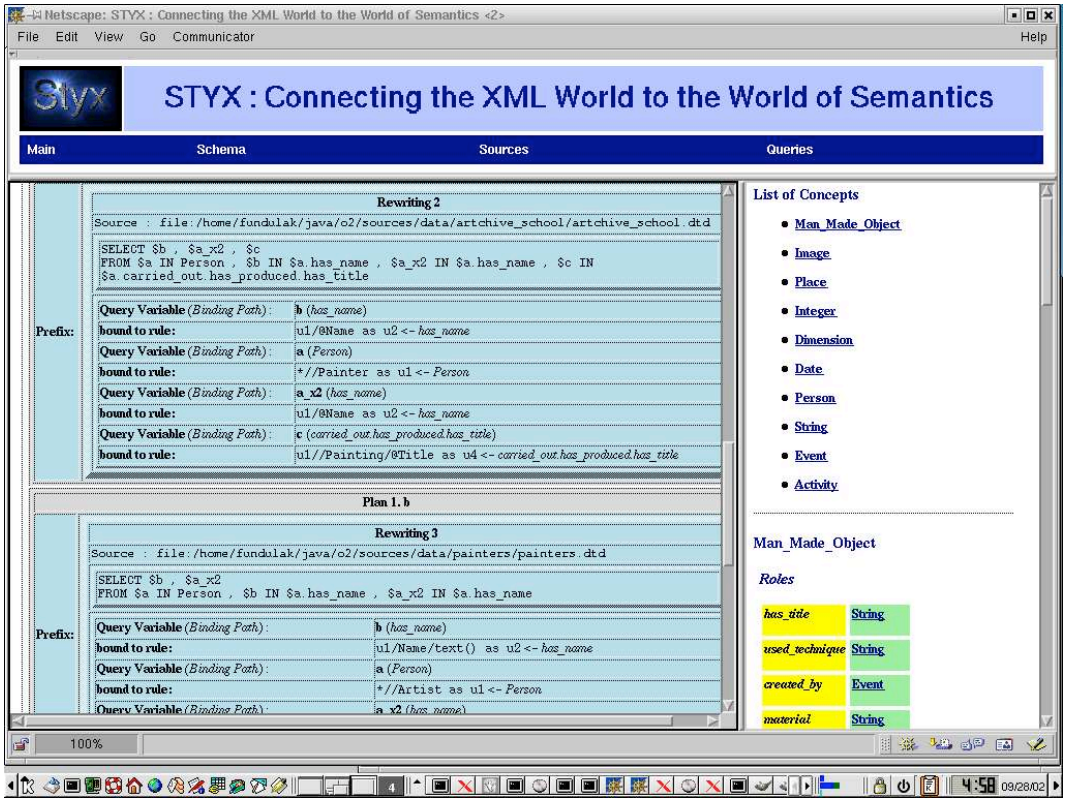

Figure 4.24: Query Execution Plan

Chapter 5

Conclusions and Future Work

In this thesis we have presented a framework for the deployment of Web community portals. We have worked on the following issues :

- the definition of a methodology for the creation of *portal schemas* which can be used as *metadata schemas* and as *mediator schemas*;
- the *publication* of Web resources in terms of their *content descriptive metadata* and
- finally the *querying* and *integration* of *autonomous* and *heterogeneous* XML resources.

5.1 Creation of Portal Schemas

Our methodology for the construction of portal schemas was based on the integration of *ontologies* and *thesauri*. Ontologies describe the basic notions of a domain or application of interest and are defined by specialists in the domain after some common agreement. Thesauri (which can be considered as a special kind of application ontologies) consist of hundreds of thousands of terms and have been extensively used as efficient means for *consistent indexing* and *retrieval* of information in several domains. Thesaurus terms are structured using a set of *predefined* relationships.

Although thesauri contain large hierarchies of terms, there is no way to express relationships between the latter (other than the predefined ones). On the other hand, in an ontology one is able to define *rich semantic relationships* between concepts to structure information as necessary. But, ontologies come with usually shallow hierarchies of concepts. In this sense, ontologies and thesauri can be considered as orthogonal ways for describing information and the former can be used as a structural interface over thesauri. This observation lead to us define a methodology for the construction of portal schemas that incorporate the *structural view of the ontologies* and the *deep hierarchical taxonomies provided by thesauri* by the integration of these information sources. Our method is independent of the format in which the resulting portal schema is represented.

The result of this research work was validated by two projects : the European C-Web project and a national project between the Conservatoire National des Arts et Métiers and the French Ministry of Culture. During these projects we have collaborated with users who have very easily identified the need for such a methodology. The ELIOT portal presented in Section 3.5 was installed in the services of the division of *Patrimoine et Archaelogie* of the French Ministry of culture and used by the archaeologists to create and query metadata descriptions of HTML documents.

In our methodology, we have considered thesauri constructed using only the *hierarchical* relationship *broader-term generic* which carries subset semantics. An interesting perspective is to study

how the other thesaurus relationships (equivalence and associative) presented in Section 2.1.2 could be taken under consideration and to see how these relationships could be used for the creation of metadata and more specifically for query processing.

5.2 $STyX$: Ontology-based Integration of XML Web resources

In the second part of the thesis we have worked on the problem of *querying heterogeneous* and *autonomous* XML resources in a domain of interest, using a *mediator-based* architecture and following the *local as view* approach to describe sources to the mediator. The result of this work was the $STyX$ prototype. The novelty of our approach consists in the *description of hierarchical XML resources in terms of general conceptual graphs* with *inheritance* relationships (*ontology*). In contrast to XML data integration systems which use either a relational view [149] or an XML view [55] of data, the presence of a rich schema at the mediator allows one to describe a large variety of XML resources in the desired level of detail. Moreover, the presence of *symmetric roles* in the ontology allows one to formulate (i) mapping rules and (ii) queries in a natural way.

XML resources in $STyX$ are published by *mapping rules* which associate *XPath location paths* with *ontology paths*. The mapping language allows one to both associate *explicitly ontology concepts* with *XML fragments* in the sources, and to attribute *specific semantics* to the *parent/child* relationship between XML fragments using the *ontology paths*.

The use of XPath offers a number of advantages. First, one is able to express any kind of XML structures using the available axes of XPath (ancestor, descendant, parent, child, attribute). Second, it is possible to express information that does not exist in the XML DTD but in the actual XML document.

Query processing in $STyX$ is done as follows : the user queries the information sources by formulating *simple tree queries* in terms of the ontology. To obtain answers for a query, this must be evaluated against all sources published in $STyX$. To evaluate a query over an XML resource, it must be *rewritten* into an XML query (e.g. XQuery) expressed in terms of the local source's schema. The $STyX$ query rewriting algorithm tries to find the mapping rules that provide answers to the query by matching the *query variables binding paths* (expressed in terms of the ontology) with the *ontology paths* of the rules. In the case where during rewriting not all query variables can be mapped to a mapping rule, in contrast to [55], we attempt to complete these partial answers by *decomposing* the query. The result of this decomposition is (i) the query that the source can answer and (ii) a set of subqueries that can be possibly answered by the other sources. The partial answers obtained from the sources are then *joined* at the mediator site. This is a recursive process and stops when all sources have been considered.

In order to perform the joins between partial answers we have introduced *keys*. Two types of keys are defined : *local keys* which are used to decide whether two XML fragments that originate from the *same* source correspond to the same XML fragment, and *global* or *semantic keys* which are used to identify XML fragments that originate from different sources. Local keys in the form of XML attributes of type `ID` or the position of fragments in the document are used to identify those originating from the same source. But these are mostly *internal pointers*. On the contrary, global keys provide a common way to identify XML fragments that originate from different or even the same source.

To conclude, we have defined and implemented a *query rewriting algorithm* for *tree queries*. The introduction of *keys* in the model enables us to handle the cases where full answers from a source cannot be found. In this case, it is possible to *decompose* the query and to complete the partial answers obtained by the other sources.

5.2.1 Future Work

Extending the ontology language An interesting direction of work consists of *extending the ontology path language*. We have seen in Section 4.2 that the language is based on the *composition* of *ontology concepts*, *roles* and *attributes*. But, in the current version of the language one cannot 'narrow down' the target of a *role* or a *role path*. For example, one can specify the ontology path of mapping rule R_1 but not of R_2, both illustrated below.

R_1:	`URL/Painter/Painting` as u_1	$\rightarrow$	*Person.carried_out.produced*
R_2:	`URL/Painter/Painting` as u_1	$\rightarrow$	*Person.carried_out.produced.Painting*

The first rule states that the `Painting` elements obtained from evaluating source path `Painter/Painting` on URL are instances of concept Man_Made_Object (the *target* of the role path *carried_out.produced* in the ontology). The second rule is more precise than the first : it states that the obtained `Painting` elements are instances of concept Painting (subconcept of Man_Made_Object) and not objects in general. Consequently, adding this information in the ontology path language allows one to specify more precisely the semantics of the underlying sources.

Query rewriting must be though reconsidered : recall that the algorithm matches the binding path of query variables (except the root's) with the ontology path of the rules. If we would have allowed the use of concepts in the role paths, simple matching of paths would not have been sufficient : more complex reasoning on the *subsumption* relationship between *role paths* would be necessary. Nevertheless, the idea behind the rewriting algorithm does not change.

Another extension of the mapping language would be to consider *conditions* à la Information Manifold [136] in the definition of the rules. For example, imagine a source which contains persons born after 1900. In this case, one could specify that the value of the attribute *year* is greater than or equal to 1900. The presence of such conditions would force an additional satisfiability check after query rewriting.

Complexity & Completeness Let us discuss the complexity and completeness issues associated with the query rewriting algorithm in $STyX$. The algorithm makes use of the closure $M^\star$ of a mapping M. The size of $M^\star$ is *exponential* in the size of M. Given a query Q and a mapping M, the rewriting algorithm is *polynomial* in the size of Q and $M^\star$. This is due to the fact that the algorithm discovers bindings by examining *all legal combinations of mapping rules*.

Nevertheless, the rewriting algorithm *is not complete*. First, let us define the notion of completeness in our context : let Q be a query expressed in terms of the $STyX$ ontology $\mathcal{O}$ and let $\mathcal{D}$ be a database, instance of $\mathcal{O}$. Let $\{s_1, s_2, \ldots s_n\}$ be the set of sources published to the $STyX$ mediator. The problem of completeness can be then formulated as follows : let Q be a query expressed in terms of the ontology $\mathcal{O}$, and Q' an *S-query rewriting*. Is the set of answers for Q when evaluated against $\mathcal{D}$, the same as the set of answers of Q' when evaluated against the sources?
The answer in our case is no. There are several reasons for this incompleteness :

First case : Consider the following two sources s_1 and s_2 which are published by the set of mapping rules illustrated below.

s_1	A_1:	URL_1/`Artist` as u_1	$\rightarrow$	Person
	A_2:	u_1/`Action` as u_2	$\rightarrow$	carried_out
s_2	B_1:	URL_2/`Action` as v_1	$\rightarrow$	Activity
	B_2:	v_1/`Result` as v_2	$\rightarrow$	produced

Source s_1 contains persons (rule A_1) and the activities that these have carried out (rule A_2). Source s_2 contains activities (rule B_1) and the objects produced by these activities (rule B_2). The first source returns *persons* and *activities* and the second returns *activities* and *man made objects*.

Consider now query Q in Table 5.1 which looks for *"the objects created by persons"* :

Q:	**select**	x_2	Q'	**select**	x_2
	from	Person x_1,		**from**	Person x_1,
		x_1.carried_out.produced x_2			x_1.carried_out a,
					a.produced x_2

Table 5.1: Queries Q and Q'

When the rewriting algorithm considers source s_1, it will associate variable x_1 with rule A_1. Variable x_2 is not associated with rule A_2, since the ontology path of the rule is not the same as x_2's binding path. Source s_2 will not be considered for rewriting, since there is no absolute rule whose ontology path is the same, or is subsumed by x_1's binding path. Nevertheless, if the query formulated was query Q' illustrated in Table 5.1 then, when rewriting for the first source, the partial binding $[x_1 \mapsto A_1, a \mapsto A_2]$ would have been created. Given this binding, the query would be decomposed into the prefix query requesting *"the activities of persons"* and the suffix query looking for *"the activities and the objects produced"* for which a full binding can be found by the second source.

The above example makes evident that in order not to miss any answers for a query, this must be reformulated into a query where all query variable binding paths with length greater than 1, must be decomposed into compositions of paths of length 1. In this case, new 'weakly' quantified existential variables must be introduced. In other words, out of the initial query, a set of queries subsumed by the initial one must be produced and each of them must be considered for evaluation.

Another solution to this problem would be to calculate the closure of *the set of mappings for all sources* published in $STyX$. In other words the *joins* between the sources are precomputed and the rewriting is performed using this closure. Nevertheless, the major disadvantage of this solution is that it is not flexible in the case of the modification of a source which will lead to the re-computation of the closure.

Second Case : Another source of incompleteness is found in the use of *keys* to perform joins between partial results. Consider for example query Q illustrated below which looks for *"the title of man made objects and their year of creation"*.

Q	**select**	x_2, x_3
	from	Man_Made_Object x_1
		x_1.has_title x_2,
		x_1.created_by.took_place_in.year x_3

Consider the two sources s_3 and s_4 published by the mapping rules illustrated below.

s_3	C_1:	URL_3/`Artifact` as u_1	$\rightarrow$	Man_Made_Object
	C_2:	u_1/`@Title` as u_2	$\rightarrow$	has_title
s_4	D_1:	URL_4/`Cultural_Object` as v_1	$\rightarrow$	Man_Made_Object
	D_2:	v_1/`@Year` as v_2	$\rightarrow$	created_by.took_place_in.year

When the rewriting algorithm considers source s_3, it creates binding $[x_1 \mapsto C_1, x_2 \mapsto C_2]$. Given the binding, the query is decomposed into a prefix and a suffix query, both illustrated below.

Q_p	**select**	x_2	Q_s	**select**	x_3
	from	Man_Made_Object x_1		**from**	Man_Made_Object x_1
		x_1.has_title x_2			x_1.created_by.took_place_in.year x_3

When the partial results are obtained, a join on the instances of variable x_1 (instances of concept Man_Made_Object) must be performed. Consider that concept Man_Made_Object is associated with two keys : k_1 and k_2. Source s_3 provides values for key k_1 and source s_4 for values of key k_2. Hence, it is not possible to perform *any* join between instances of x_1 obtained from the two sources, since these do not have a common key for these instances.
Suppose now a third source s_5 which returns values for both keys. When computing the integrated database $\mathcal{D}$, the presence of this source allows to join the objects obtained from the partial databases of sources s_3 and s_4. But, this source is not considered in the rewriting. A solution to this problem would be to use source s_5 in the final plan and the key values obtained could be used to perform the necessary joins. More general, the idea is to calculate the *transitive closure* of keys and to use this information in the plan.

Another direction of research concerns the optimization of the final query execution plan which is a union of joins of prefix queries. It might be the case that in such a join, there might exist prefix queries which will be sent by evaluation to the same source. An optimization that could be considered is the *composition* of queries to be evaluated on the same source instead of evaluating them separately and then perform a join on the mediator site (i.e. pushing joins to the sources). Last, an issue that should be studied is to extend the query rewriting algorithm to handle *graph queries*.

Appendix A

XML

The **eXtensible Markup Language** or **XML** [60] is a language proposed as a standard for the representation and exchange of Web documents proposed by the World Wide Web Consortium [215]. XML is proposed by the W3C as a language for the *exchange* and *meaningful representation* of Web documents. It is a descendant of SGML [98] and of HTML[1].

The **World Wide Web (Web)** has emerged during the last decade as the central forum for data storage and exchange, and as the infrastructure for a large part of human communications and information based activities in many domains. The principle of the Web is that information is exchanged in the form of files, that are uniquely identified by a *Uniform Resource Locator (URL)*. Information can be either a simple text file, an image, a sound or a video i.e. it has a minimal structure. To represent information on the Web, HTML (Hypertext Markup Language) is used, which is the lingua franca for publishing hypertext information on the Web. It is a non-proprietary format based upon SGML, and can be created and processed by a wide range of tools, from simple plain text editors to sophisticated WYSIWYG authoring tools. HTML, as in SGML, is based on the principle of *tags*, special elements in the HTML documents, that dictate how the document should be visualized by the navigators.

Currently, data exchanged on the Web is not just static HTML pages but also data that is dynamically exported by databases. originates from databases. Moreover, the applications that are accessing these pages are not limited to the navigator. Consider a travel agency that cooperates with many hotels, advertising them on the Web by means of HTML pages. More specifically, these pages are created on demand (by an SQL query) that accesses a relational database where all data relevant to those hotels is stored. The result of this SQL query is then appropriately formatted into a collection of HTML pages.
Consider now another travel agency that needs to get this information to offer to its clients the best prices. The only way to access this data is through the HTML pages provided by the first agent. For that, a specific software needs to be written that parses these pages and extracts the necessary data. The problem here is that any modification of their formatting would cause the complete rewriting of the software. It could also be the case that if the second agent needs for example to access the average price of hotels in Greece, then *all* HTML pages should be downloaded, and a rather complicated software should be written to get this information (even though this is implemented by a simple SQL query invoked on a single column of the relational database).

This simple example demonstrates the need for the definition of *new formats* for the *representation* and *exchange* of Web. XML (eXtensible Markup Language) [4] has emerged as the *de-facto*

[1] http://www.w3.org/MarkUp/

standard for the representation and exchange of Web data. It was specifically designed to describe *content* rather than *presentation*, as is the case with HTML. It is a descendant of SGML and it complements HTML. It is based on special *elements* or *tags* which, on the contrary with HTML, are *semantic* ones : they are there to give precise meaning to the content that they englobe and they do not contain any information on how the contents could be visualized by a navigator. Differences between HTML and XML can be summarized as follows :

1. HTML tags concern *only* the visual representation of the document. On the other hand, XML tags are *semantic* tags that capture the document's real structure;
2. new semantic tags can be *added* at any time in an XML document to indicate new content;
3. the XML document's structure can be arbitrarily complex;
4. an XML document can contain the description of its *grammar (schema)*, known as *Document Type Definition*;

A.1 XML Syntax

An XML document consists of three parts :

1. an *XML declaration*;
2. a *Document Type Definition* and finally
3. the *document's content*.

A.1.1 XML Declaration

An example of an XML document is shown in Figure A.1. The document starts with the *XML declaration* (line 1)

```
<?xml version=''1.0''?>
```

This declaration appears in the beginning of every XML document and indicates the *version* of XML that this document is written in. Although this declaration is an optional one, it is necessary to appear since it is used by parsers and other tools to analyze the document with respect to the XML version that is written in. At the same time, we could also associate some attributes to this declaration such as the character encoding used in the document.

A.1.2 Declaration of the Document Type

The second part of an XML document defines the XML *DTD* (*Document Type Definition*) that describes the *grammar* or *general structure* of the document. In the example XML document in Figure A.1 this declaration is found between lines 2 and 12. This description can either be included in the document (the case for the document in Figure A.1) or can be declared externally and specified by using a declaration of the type :

```
<!DOCTYPE W4F_DOC SYSTEM ‘‘w4f_doc.dtd’’ [local declarations]>
```

The filename specified after the `SYSTEM` keyword can be either a *local file name* or a *URL*. The usage of URLs allows multiple documents (that might reside on different servers) to share the same DTD. The part of local declarations, allows one to define new structures or to modify existing ones. This is done *only* for the XML document in the context of which these modifications are done and do not affect the XML documents that possibly share the same XML DTD.

The declaration of an XML DTD for an XML document is optional. If it appears, it should be before the document content and the structure of the document must respect the structures defined by the DTD.

```
1.<?xml version=''1.0''?>
2. <!DOCTYPE W4F_DOC SYSTEM ``w4f_doc.dtd'' [
3.   <!ELEMENT W4F_DOC (Painter | Painting)*>
4.   <!ELEMENT Painter (Paintings_list)>
5.   <!ATTLIST Painter Name CDATA #IMPLIED>
6.   <!ELEMENT Paintings_list (Painting)*>
7.   <!ELEMENT Painting (Year, Technique?)>
8.   <!ATTLIST Painting Title CDATA #IMPLIED>
9.   <!ELEMENT Year (#PCDATA)>
10.  <!ELEMENT Technique (#PCDATA)>
11.  <!ENTITY  Comment ``Private Collection''>
12.  <!ENTITY  Biography SYSTEM ``biography.xml''>]
13.  <W4F_DOC>
14.    <Painter Name='Georges Braque'>
15.      &Biography;
16.      <Paintings_list>
17.        <Painting Title="Black Fish">
18.          <Year>1942</Year>
19.          <Technique>Oil on canvas</Technique>
20.          <COMMENT>&Comment;</COMMENT>
21.        </Painting>
22.        <Painting Title="Bottle and Fishes">
23.          <Year>autumn 1910</Year>
24.          <Technique>Oil on canvas</Technique>
25.        </Painting>
26.     </Paintings_list>
27.    </Painter>
28.  </W4F_DOC>
```

Figure A.1: An XML document

A.1.3 XML Entities

The use of *entities* is a mean to factorize parts of an XML document and to reuse them in several places. An entity is *always* declared in an XML DTD.
In the document shown in Figure A.1, the declarations are local ones (the XML DTD is defined locally) but they can be part of an external DTD. The XML DTD defined in the document of Figure A.1, the entities `Comment` and `Biography` (lines `11` and `12` respectively) are defined. The first is an *internal entity* (i.e it is defined in the XML document) and the second is an *external one* (an XML file). An entity is used in an XML document by a reference of the form `&name;` where *name* is the entity's name (for example the use of entity `Comment` in line `20`). .

A.1.4 XML Elements

The basic component of an XML document is an *element*. Each element is composed of (i) a *start-tag* (e.g. $< type >$), (ii) its *content* and (iii) an *end-tag* (e.g. $< /type >$) where *type* is the *element name* and it corresponds to the *element type*. For example, in the previous document there exist seven elements : `W4F_DOC`, `Painter`, `Paintings_list`, `Painting`, `Year`, `Technique`, `COMMENT`, as well as all the elements defined in the document *biography.xml*. From this point on we use the terms *element* and *element type* interchangeably.

The names of elements that might appear in an XML document are free and are defined using the letters of the alphabet, numbers and the characters `'-'` and `'_'`. XML is case-sensitive, so the element name `Painter` is not the same as `PAINTER`. Moreover, element names cannot (i) start by a number and (ii) contain blanks.

Each XML document has a single element that serves as the *root* of the XML document, which defines the content of the document itself. For the XML document illustrated in Figure A.1, the root element is of type `W4F_DOC`. If the document is associated to a DTD, the type of the root element must be the same as the type identified after the declaration `DOCTYPE`.

The *content* of an XML element is determined by whatever is specified *between the element's start-tag and end-tag* : other elements, entities, text, comments, processing instructions etc. For our example document, the content of the element of `Painter` (line `14`) consists of : an element of type `Paintings_list` (line `16`) and a reference to the entity `Biography` (line `15`). The element of type `Paintings_list` contains in its turn two elements of type `Painting` (lines `17, 22`) where each one of them contains an element of type `Year` (lines `18, 23`) and an element of type `Technique` (lines `19, 24`). The first `Painting` element (line `17`) contains also an element of type `COMMENT` (line `20`). An element can appear more than one times in the content of another element. This is the case of the `Painting` elements in the previous example. The content of an XML element can be empty.

Tags in XML are user-defined, in contrast to HTML [214] where they are defined by HTML syntax. An important issue concerning XML documents is that they must be *well-formed* :

1. for each start-tag there should be an end-tag (and vise versa) and
2. end-tags should appear in *inverse order* that their corresponding start-tags have been declared.

An example of an XML document which is not well formed is illustrated in Figure A.2. In this document, the end-tag of the XML element `Painter` does not exist in the document (line `17`). Remark also that the order of the end-tags of elements `Painting` and `Technique` (lines `10`) has changed. More specifically, the end-tag of element `Painting` appears *before* the end-tag of element `Technique` although their start-tags appear in the inverse order. From this point on we assume that the XML documents that we deal with are well-formed.

A.1.5 XML Attributes

Each XML element can be associated with one or more *attributes*. XML attributes can be essentially considered as *properties* of the XML element. XML attributes are defined as couples of *(name, value)* pairs where *name* is the attribute name and *value* the attribute value: an XML attribute has a *value* but not a *content* as is the case for XML elements.
For the XML document illustrated in Figure A.1, the element `Painter` (line `14`) is associated with the attribute `Name` with value *'Georges Braque'* that indicates the name of the painter.

As with XML elements, XML attributes are user defined. Principles for attribute names are similar to those for element names. The value of an XML attribute is always a sequence of characters

```
1.<?xml version=''1.0''?>
2.    <Painter Name='Georges Braque'>
3.      &Biography;
4.      <Paintings_list>
5.        <Painting Title="Black Fish">
6.          <Year>1942</Year>
7.          <Technique>Oil on canvas</Technique>
8.          <COMMENT>&Comment;</COMMENT>
9.       </Painting>
10.        <Painting Title="Bottle and Fishes">
11.          <Year>autumn 1910</Year>
12.          <Technique>Oil on canvas
13.       </Painting>
14.          </Technique>
15.        </Painting>
16.     </Paintings_list>
17.  </W4F_DOC>
```

Figure A.2: A non well-formed XML document

(string). An attribute of an element in a well-formed XML document can occur only *once* within the context of the element. An attribute is *always* associated to a value.

A.2 XML : Document Type Definitions

As illustrated previously, an XML document *might* be associated to a *document type definition (DTD)* which serves as the *grammar* that the XML document must respect. An XML DTD can also be seen as a *schema*, from the database perspective, that an XML document conforming to this DTD (instance) should respect.

In the XML DTD, *element types* with their *content models* are defined, as well as well all the entities that will be eventually used by documents conforming to this DTD. Moreover, to each element type we can also associate XML *attributes*.

The presence of an XML DTD offers a number of advantages for the exploitation of XML documents : document production through XML editors and XSLT programming [82]. One of the most important advantages is the use of DTDs for the *exchange* of data. In a producer/consumer context, the presence of DTDs facilitates the exchange of data from the consumer point of view. If the consumer knows the structure of the XML documents that he will receive from the producer, the construction of the necessary software to process those documents becomes an easier task . . Nevertheless, there are several limitations concerning XML DTDs:

- an XML DTD is not an XML document, consequently tools for processing XML documents cannot be used to process DTDs (parsers, stylesheets processors, etc.);
- there is no typing for element content;
- enforcing a restriction on the number of child elements is not possible (for example one cannot restrict the number of `Painting` elements to be *at-most* 3);
- and finally there is no subtyping between element types. Consequently it is not possibly to share structures between different elements.

Example A.2.1 *Consider the XML DTD illustrated in Figure A.1. The element of type* `W4F_DOC` *is defined to be the root element of the XML document (line* `2`*). An element of this type (line* `3`*) consists of zero or more (occurrence indicator "*") elements of type* `Painter`*, or elements of type* `Painting`*. An element of type* `Painter` *(line* `4`*) consists of an element of type* `Paintings_list`*. An element of this type consists of zero or more (occurrence indicator "*") elements of type* `Painting`*. An element of type* `Painting` *(line* `7`*) consists of (aggregation connector ",") an element of type* `Year`*, and an optional element of type* `Technique` *(occurrence indicator "?").*

Last, an element of type `Painter`*, is associated to an XML attribute* `Name` *(the name of the painter) (line* `5`*) and an element of type* `Painting` *is associated to an XML attribute* `Title` *(the title of the painting) (line* `8`*).*

A.2.1 Element Declaration

An XML DTD defines for each *element type*, its *name*, its *structure* and *attributes*. The definition of an XML element in a DTD is of the form :

```
<!ELEMENT element\_name content\_model>
```

The keyword `ELEMENT` indicates the element declaration. It is followed by the element's name (`element_name`) and then by the description of the structure of the element (`content_model`). There are 5 different types of element content models :

1. empty;
2. it might contain only text, a number of sub-elements, contain text and sub-elements (mixed content model)
3. and last it can contain *any* combination of other elements.

A content model for an XML element indicates not only the *type* of its sub-elements but also (i) the *order* in which they should appear and (ii) the number of occurrences of its sub-elements. More specifically, XML defines the following occurrence indicators :

1. "*" for zero or more,
2. "+" for one or more and last
3. "?" for zero or one occurrence of an element.

At the same time, XML forces order of elements with the *aggregation operator* "," or no order using the *alternation* operator "|".

More specifically, if e is an element type, then the content of an element is determined by the regular expressions $e*$ (zero or more occurrences), $e+$ (one or more occurrences), $e \mid e'$ (alternation), $e?$ (zero or one) and e, e' (aggregation).

An XML DTD is in fact a *context-free grammar* for the XML document. For the previous example, consider the element type `Painting`. An element of this type contains an element of type `Year`, and an optional element of type `Technique` and these elements should appear in an document in this order. Since a context-free grammar can be recursive, we might have an XML document where an element can contain a sub-element of the same type.

A.2.2 Attributes Declaration

An attribute is always declared within the context of an XML element :

```
<!ATTLIST element_name attribute_name attribute_definition>
```

`element_name` is the name of the element in whose context the attribute is defined, `attribute_name` is the name of the attribute.
`attribute_definition` consists of (i) the attribute type (which can be a string, a tokenized type or an enumerated type), (ii) constraints specifying whether the attribute is mandatory or optional and (iii) possible default values for it.

Consider for example the XML DTD illustrated in Figure A.1. An element of type `Painter` is associated to an attribute `Name`, which is of type string (`CDATA`), and is mandatory (declaration `#REQUIRED`).

XML supports specific attribute types : `ID, IDREF, IDREFS`. When an attribute is declared to be of type `ID` in the context of an element, it means that the value of this attribute can be used as an *internal identifier* for all elements of this type *within an XML document* : there cannot exist two elements in the XML document which have the same value for this attribute.
When an attribute of type `IDREF` is declared in the context of an element, it means that this attribute references an element for which an attribute of type `ID` has been declared (the attribute's value is some other element's identifier declared with an attribute of type `ID`). `IDREFS` means that the value of this attribute is a list of identifiers separated by spaces.

Example A.2.2 *Consider the previous XML DTD which is now modified and illustrated in Figure A.3. Consider the XML element of type* `Painter` *which is the empty element. The attribute* `painter_id` *declared in the context of this element (line* `6`*) is of type* `ID` *and is mandatory. Similarly, we have defined for the element of type* `Painting` *the attribute* `painting_id` *of type* `ID`*, also mandatory. In the contrary to the previous version of the XML DTD where paintings of a painter are defined below their respective painter, in this DTD, the paintings of a painter are described using the attribute* `painter` *defined in the elements of type* `Painting`*: for each element of this type, the value of this attribute is an identifier already defined in the XML document. This example reveals another problem of XML DTDs. Here, references are not typed. This means, that the values used for an attribute of type* `IDREF` *cannot be typed : for example we cannot restrict the values of the attribute* `painter` *defined in* `Painting` *element type to take its values in the set of values of the* `painter_id` *attribute defined in* `Painter` *elements.*

As illustrated previously, a *well-formed* XML document is a document with balanced start and end tags. If a well-formed XML document has a DTD, and in addition conforms to this DTD, then it is called *valid* : it respects (i) the grammar of the DTD and (ii) the definitions concerning the structure of the elements (attributes).

An example of an XML document which is not valid is shown in Figure A.4. Remark that the `Name` subelement appears *before* the `Painting` subelement in the definition of the of the `Painter` element (line `4`). But in the XML document this order is inversed (lines `12` and `17` respectively). Also, a `Painting` element can have either one or zero `Year` subelements. But in the XML document, the `Painting` element has two such subelements. The document does not respect the XML DTD and consequently is not valid.

```
1.<?xml version=''1.0''?>
2. <!DOCTYPE W4F_DOC SYSTEM ``w4f_doc.dtd'' [
3.   <!ELEMENT W4F_DOC (Painter | Painting)*>
4.   <!ELEMENT Painter EMPTY>
5.   <!ATTLIST Painter Name CDATA #IMPLIED
6.                     painter_id ID #REQUIRED>
7.   <!ELEMENT Painting (Year, Technique?)>
8.   <!ATTLIST Painting Title CDATA #IMPLIED
9                       painting_id ID #IMPLIED
10.                     painter IDREF #REQUIRED>
11.   <!ELEMENT Year (#PCDATA)>
12.  <!ELEMENT Technique (#PCDATA)>
13.  <!ENTITY  Comment ``Private Collection''>
14.  <!ENTITY  Biography SYSTEM ``biography.xml''>]
15.  <W4F_DOC>
16.    <Painter Name='Georges Braque' painter_id='1'>
17.      &Biography;
18.    </Painter>
19.    <Painting Title="Black Fish" painter='1'>
20.      <Year>1942</Year>
21.      <Technique>Oil on canvas</Technique>
22.      <COMMENT>&Comment;</COMMENT>
23.    </Painting>
24.    <Painting Title="Bottle and Fishes" painter_id='1'>
25.      <Year>autumn 1910</Year>
26.      <Technique>Oil on canvas</Technique>
27.    </Painting>
28.  </W4F_DOC>
```

Figure A.3: Another XML document

```
1.<?xml version=''1.0''?>
2. <!DOCTYPE W4F_DOC SYSTEM ``w4f_doc.dtd'' [
3.   <!ELEMENT W4F_DOC (Painter)*>
4.   <!ELEMENT Painter (Name, Painting)>
5.   <!ELEMENT Painting (Technique, Year?)>
6.   <!ATTLIST Painting Title CDATA #REQUIRED>
7.   <!ELEMENT Year (#PCDATA)>
8.   <!ELEMENT Technique (#PCDATA)>
9.   <!ELEMENT Technique (#PCDATA)>
10.  <W4F_DOC>
11.    <Painter>
12.      <Painting>
13.        <Year>1942</Year>
14.        <Year>1943</Year>
15.        <Technique>Oil on canvas</Technique>
16.      </Painting>
17.      <Name>'Georges Braque'</Name>
18.    </Painter>
19.  </W4F_DOC>
```

Figure A.4: A non valid XML document

Appendix B

XML Path Language (XPath)

The XML Path Language (XPath) [52] is a W3C Recommendation whose main purpose is to address parts of an XML document. It is a *non-XML* language and it provides a common syntax and semantics for the functionality shared between XSLT [82] and XPointer [186]. Moreover, the significance of XPath language is considerable since it is the language used by the majority of XML query languages for binding query variables.

XPath works on the XML document considering it as a *tree* of *nodes*. XPath indicates nodes in the document tree by their *absolute/relative position*, *type*, *content* and other more complex criteria. XPath allows to *navigate* in the document tree, as well as to *select* tree nodes that fulfill a number of conditions.

The primary syntactic construct in XPath is a *path expression*, which is always evaluated with respect to a *context*. In the following we present first the tree model of XPath, and then the basic XPath expressions.

B.1 XPath Data Model

An XML document can be seen as a *tree*, which consists of *nodes* of different types that correspond to the different syntactic XML categories. XML parsers such as those based on the model DOM (*Document Object Model*) [213], construct a tree before applying any treatment to the document. The tree representation can be seen as a *conceptual* representation of the document, that separates the syntax from a more abstract representation of the document content. In an XML tree there are essentially seven types of nodes, illustrated in Table B.1 each one of them is associated to a syntactic category of XML.

Node Types	Syntactic Category of XML
Document	XML document
Element	XML element
Attr	XML attribute
Text	text values
Comment	XML comments
ProcessingInstruction	processing instructions
Namespace	namespaces

Table B.1: XML node types in an XML tree

In the XPath data model the *root node* of the XML document is not the same as the *root element*. The root node of the XML document is of type Document, it appears always first and contains the entire document. More precisely, it contains the node corresponding to the root element, and any comments and processing instructions that occur *outside* the root element.

One can observe that in the previous set of node types there is no type neither for the DTD nor for the entities. In fact the tree model does not consider the presence of a DTD, and where the references toward the entities have been replaced by their textual content.

For each node type there exist two types of information: first is the *name* of the node and the second its *value* (Table B.2).

Node Type	Name	Value
Document	–	–
Element	element name	–
Attr	attribute name	attribute value
Text	–	text
Comment	–	comment's text
Namespace	prefix	URI that designates the namespace

Table B.2: XML node types : their name and value

The XML tree for the XML document illustrated in Figure A.1 is shown in Figure B.1.

For every type of node, there is a way of determining the *string-value* of nodes of that type. For some node types, the string-value is part of the node; for other, the string-value is computed from the string-value of descendant nodes.

Some node types have also an *expanded-name*. This name consists of a *local part* and from a *namespace URI*. The local part is a string and the namespace URI is either a string or null. For example, for the element <Painter> its local name is *Painter* and its namespace URI is null. For the element <xsd:Painting> the expanded name is *xsd:Painting* where the local part is *"Painting"* and the namespace URI is *"xsd"* defined in the document.

Moreover, in an XML tree there is an *ordering*, defined on all the nodes in the tree. This order corresponds to the order in which the first character of the XML representation of each node appears in the XML representation of the document after expansion of the entities.

Element nodes have a (possible empty) *list* of *child* nodes. The root node has at least one child node : the document root element. The root node of the XML document is of type Document and it always appears first. It contains a *single* child of type Element which corresponds to the root element of the document. Each node of type Element can have a number of *children nodes*. Namespace nodes appear before attribute nodes and both appear before the children nodes of the element.

Example B.1.1 *For the document illustrated in Figure A.1, Page 157, the root of the tree contains the root element* W4F_DOC. *This element has one child node of type* Element, *whose name is* Painter. *In turn, this one has one child node of type* Element *whose name is* Paintings_list. *This node has two children, both of type* Element. *The first, has four child nodes: the first is of type* Attribute, *its name is* Title *and its value is "Black Fish"; the other three are of type* Element. *The name of the first is* Year, *of the second* Technique *and of the third* Comment. *All three have only one child node of type* Text.

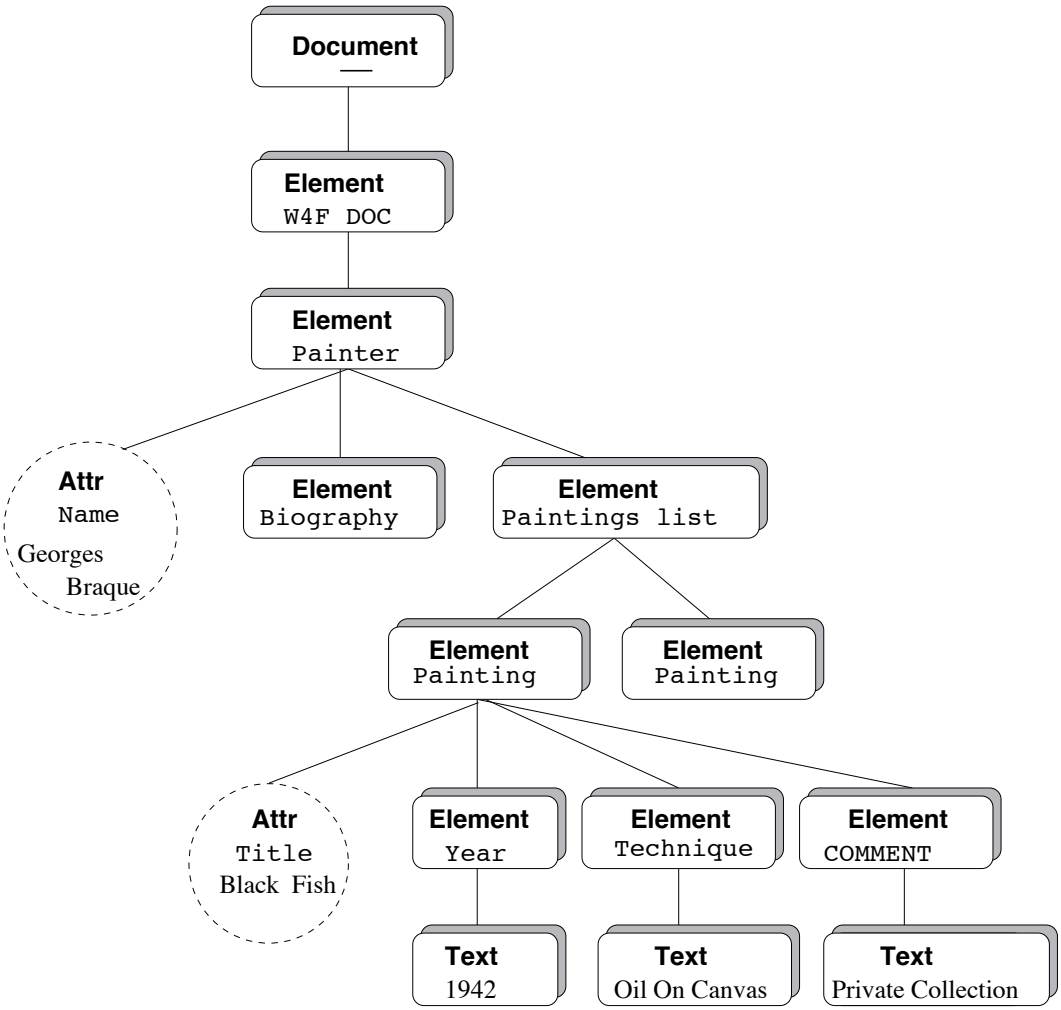

Figure B.1: The XML Tree for the XML document in Figure A.1

B.1.1 XPath expressions

In XPath, the referencing of nodes is done by *navigating* in the tree. Navigation is done by means of *path expressions* always evaluated in a *context*. XPath expressions are either *absolute* or *relative*. In this first case, the expression is evaluated on the *root* node of the XML tree, while in the second case is evaluate on some *context* (calculated by another path expression).

XPath expressions are called *location paths*. A location path is a sequence of *location steps*. Each step is evaluated in some *context*, which is a set of XML nodes defined by the previous location step if it is a relative path, or on the document root element if it is an absolute path. The general form of an XPath location path is :

$$[/]step_1/step_2/\ldots/step_n$$

where "/" is the initial optional step that defines whether the path is an absolute one or a relative one.

The general form of a location step is

$$\text{axis::node\_test}[predicate_1][predicate_2]\ldots[predicate_n]$$

and 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 nodes that will be considered in the result set of the expression. The role of the node test is dual: it can either specify the type of the node (element, attribute, comment, processing instruction etc.) and/or the name of the node in the XML tree,
3. and *optional predicates*, which use arbitrary expressions to further refine the set of nodes selected by the location step.

The result of an XPath expression can be (i) a set of nodes (an unordered collection of nodes without duplicates called *node-set* in the XPath jargon), (ii) a *boolean* (true or false), (iii) a *number* and (iv) a *string* (a sequence of characters). The members of a node-set are not the actual nodes in the XML tree but rather *references* to these nodes.

In order to evaluate a location path, the location steps are evaluated in the order they are defined in the path : on the result obtained by the evaluation of $step_i$, $step_{i+1}$ is evaluated. Location paths can be used in two different ways. First, a location path can be considered as a pattern for *selecting* XML fragments. Second, it is a way to *navigate* in the XML document from node to node.

XPath supports a number of different axis for the navigation in the XML document. These allow the navigation in the XML document from node to node. XPath axes are illustrated in Table B.3 and they can be classified into mainly three categories :

- Axes child, descendant, descendant-or-self are used for navigating *downwards* in the XML document.
- Axes parent, ancestor, ancestor-or-self for *backward* navigation.
- Axes preceding-sibling, following-sibling, preceding and following for *sideway* navigation.
- Finally, axe attribute is used for selecting *attribute* nodes.

Each axis has a *principle node type*. More specifically :

Axe	Description
child	selects the child node of the context node
descendant	selects the descendant nodes of the context node
descendant-or-self	selects the context node and all its descendant nodes
parent	selects the parent node of the context node
ancestor	selects all ancestor nodes of the context node
ancestor-or-self	selects the context node and all its ancestor nodes
preceding-sibling	selects the sibling node that precedes the context node
following-sibling	selects the sibling node that follows the context node
preceding	selects all sibling nodes preceding the context node
following	selects all sibling nodes following the context node
self	selects the context node
attribute	selects the attribute nodes of the context node
namespace	selects the namespace nodes

Table B.3: XPath Axis

- for the attribute axis, the principle node type is Attr;
- for the namespace axis, the principle node type is Namespace;
- and finally for all other axis, the principle node type is Element.

The node-test can be either the expanded-name of a node, or *, text(), node(). Node test *, is true for any node of the principle node type of the axis in the step. Node test text() is true for any textual node and finally, node test node() is true for *any* node.

Example B.1.2 *The absolute location path* /child::W4F_DOC/child::Painter *is composed of two location steps:*

Step /child::W4F_DOC *: This step selects the root node of the XML document (due to the presence of the "/" axis), and then selects the* child *elements of the node (axis* child*); from all the child nodes it selects the elements of type is* `W4F_DOC` *(document root element).*

Step child::Painter *: This step is evaluated on the result of the first step (on the element* `W4F_DOC`*). It selects the children nodes of this element, and then out of those the elements of type* `Painter` *are selected.*

The first location step is an absolute one (it is evaluated on the root node of the document), and the second is a relative one, is evaluated on the result of the first one.

Example B.1.3 *Consider the location path* /descendant::Painting/attribute::Title *This path is decomposed into two steps:*

Step /descendant::Painter *: This step selects the root node of the XML document (due to the presence of the "/" axis), and then selects the* descendant *elements of the node (axis* descendant*). Then from the set of those nodes it selects the elements of type* `Painting`*.*

Step attribute::Title *: This step is evaluated on the result of the first step (on the elements of type* Painting*). For each of those elements, it selects its* attribute *nodes (axis* attribute*) and from those it keeps those whose name is* `Title`*.*

As mentioned earlier, in XPath location steps we can have optional predicates that reduce the size of the result set. 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. A predicate can be in fact a complex expression that includes disjunction,conjunction and negation of atoms, where an atom is a location step, or a condition between a location step and a value.

Example B.1.4 *Consider the location path*

child::Painter[child::@Name='Georges Braque and child::Painting[@Title]]

This location step considers elements of type `Painter` *such that the predicate* child::@Name='Georges Braque and child::Painting[@Title] *is true :*

- *the value of the attribute* `Name` *of the context node is equal to 'Georges Braque' and*
- *the context node has a child element of type* `Painting` *which in turn has an attribute* `Title`.

B.1.2 XPath Functions

In location steps we can also use *functions* from the core library of XPath. These functions are distinguished into four sets :

Node Set Functions : Functions that belong in this set are :

- *last()* which returns the position of the last element in a context,
- *position()* that returns the position of an element in a context, *count()* which returns the number of nodes in the set of nodes passed as argument,
- *local-name()* that returns the local part of an expanded name of a node,
- *namespace-uri()* which returns the namespace URI of the expanded name of the element, and finally the
- *name()* function that returns the expanded name of the node.

For the three last functions if a node set is passed as an argument, then the function is applied on the first node of the argument set.

In this set of functions belongs the function *id()* which allows to navigate in the XML document. In fact this function selects elements by their *unique* ID. The argument to this function can be a string value; for example the step *id('1')* selects the *element* in the document which has an attribute of type `ID` with value `'1'`.

String functions : Functions that belong in this set are similar to the string functions of the C programming language. Those are *concat()*, *starts-with()*, *contains()*, *substring-before()*, *substring-after()*, *substring()*, *string-length()*. *concat()* concatenates its string arguments in the order they appear, *starts-with()* checks whether the second string argument is a prefix of the first, *substring-before()*/*substring-after()* returns the part of the first string argument that is before/after the second string argument, *string-length()* returns the length of the string passed as parameter. If the parameter is omitted, then this function is applied to the context node (which is first converted to its string value). Function *normalize-space()* removes all white spaces of the string passed as parameter. As in the case of *string-length()* if the argument is omitted, then the function is applied to the context node. XPath introduces also the function

translate() that performs case conversion. Last, the most important function is *string()* which converts the object passed as parameter to its *string-value*. If the object passed as parameter is a node set, then the first node in the set is considered and its string-value is returned. Numbers and boolean values are translated to string values.

Boolean Functions : The *boolean()* function converts the object passed as parameter to a boolean value. Function *not()* accepts as argument a boolean and returns its complement value. Functions *true()* and *false()* return values true and false respectively. Finally, function *lang()* verifies if the language of the context node (encoded using the xml:lang attribute is the same as the string passed as parameter.

Number Functions: Those functions operate on numbers or on node-sets and return numbers. Similar to the string functions, they resemble to functions in the C programming language. Functions *floor()*, *ceiling()* and *round()* operate on the number passed as parameter. Function *sum()* accepts as parameter a node-set, and returns the sum of the nodes in this set, where each of the nodes is transformed to a number. Last, the *number()* function converts its argument to a number. If the parameter passed is a node-set then it is first converted to a string (applying the *string()* function presented previously), which is then transformed to a number.

Example B.1.5 *Consider the location path*

child::Painting[position()=1]/id(@painter)/attribute::Name

the XML document illustrated in Figure A.3. This path consists of three location steps :

Step child::Painting[position()=1] *This step finds the* children *nodes of the context node (axis* child*) and keeps the first (node-set function* position()*) element of type* `Painting`.

Step id(@painter) *This step, uses the node-set function* id() *which selects elements by their* `ID` *value. Here the argument passed is the value of the attribute* `painter` *of the context node. The step returns the nodes in the XML tree whose value of attribute* `ID` *equals to the value of the* `painter` *attribute of the context node.*

Step attribute::Name *This step selects the* attribute *(*attribute *axis)* `Name` *of the context node.*

Bibliography

[1] Introduction to the Art & Architecture Thesaurus. Published on behalf of the *Getty Art History Information Program*, Oxforf University Press, New York.

[2] Art & Architecture Thesaurus. http://www.getty.edu/research/tools/vocabulary/aat.

[3] S. Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Data Transaction (ICDT)*, Delphi, Greece, 1997.

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

[5] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries (JODL 97)*, 1996. URL: ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-101.ps.gz.

[6] Serge Abiteboul and Oliver Duschka. Complexity of answering queries using materialized views. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, pages 254–263, Seattle, Washington, May 1998.

[7] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 253–262, Portland, Oregon, 1989. ACM Press.

[8] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes in Microsoft SQL Server. In *Proceedings of VLDB*, Cairo, Egypt, 2000.

[9] Vincent Aguilera. *Interrogation de documents XML*. PhD thesis, Ecole Nationale des Ponts et Chaussees, 2002.

[10] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, and Fanny Wattez. Querying XML documents in Xyleme. ACM SIGIR Workshop on XML and Information Retrieval, July 2000.

[11] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[12] 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 Int'l Workshop on The World Wide Web and Conceptual Modelling*, 2000.

[13] S. Alexaki, G. Karvounarakis, V. Christophides, D. Plexousakis, and K. Tolle. Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, pages 1–13, Hong-Kong, May 2001.

[14] Altavista web page. http://www.altavista.com.

[15] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Ontology-based Integration of XML Resources. In *Proc. of the First International Conference on the Semantic Web*, Sardinia, Italy, June 2002. LNCS.

[16] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Querying XML sources using an Ontology-based Mediator. In *Proc. of Int. Conference on Cooperative Information Systems (CoopIS)*, Irvine, California, USA, November 2002.

[17] Bernd Amann, Catriel Beeri, Irini Fundulaki, Michel Scholl, and Anne-Marie Vercoustre. Mapping XML fragments to Community Web Ontologies. Informal Proceedings of the Fourth International Workshop on Web and Databases (WebDB), May 2001. In conjunction with ACM SIGMOD-PODS.

[18] Bernd Amann and Irini Fundulaki. Integrating Ontologies and Thesauri to Build RDF Schemas. In *Proceedings of the 3rd European Conference on Research and Advanced Technology for Digital Libraries*, Paris, France, September 1998.

[19] Bernd Amann, Irini Fundulaki, and Michel Scholl. Integrating Ontologies and Thesauri for RDF schema creation and metadata querying. *International Journal of Digital Libraries*, 2000.

[20] J. Ordille A.Y. Levy, A. Rajaraman. Query Answering Algorithms for Information Agents. In *Proc. of the Thirteen International Conference on Artificial Intelligence, AAAI-96*, Portland, OH, 1996.

[21] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *Description Logic Handbook*. Cambridge University Press, 2002.

[22] C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In Exhibitions Program of ACM SIGMOD Int. Conference on Management of Data, June 1999.

[23] C. Batini, M. Lenzerini, and M. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. In *ACM Computing Surveys*, volume 18, pages 323–364. ACM Press, 1986.

[24] C. Beeri, A. Levy, and M-C. Rousset. Rewriting Queries Using Views in Description Logics. In *Proc. PODS*, pages 99–108, Tucson, Arizona, May 1997.

[25] S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Record*, 28(1):54–59, 1999.

[26] Paul V. Biron and Ashok Malhotra. XML Schema Part 2 : Datatypes. W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-2.

[27] A. Borgida. Description Logics in Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[28] A. Borgida, R. Brachman, D. McGuiness, and L. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 58–66, Portland, Oregon, June 1989.

[29] T. Bray. RDF and Metadata, June 1998. http://www.xml.com/xml/pub/98/06/rdf.html.

[30] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Technical Report CR-rdf-schema-20000327, W3C, March 2000. W3C Candidate Recommendation.

[31] P. Buneman. Semistructured data. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, Tucson, Arizona, 1997.

[32] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. In *Proc. WWW10*, pages 201–210, 2001.

[33] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[34] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of Regular Expressions and Regular Path Queries. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, pages 194–204, Philadelphia, Pennsylvania, USA, 1999.

[35] Diego Calvanese, Giusepe De Giacomo, Mauricio Lenzerini, and Moshe Vardi. Answering Regular Path Queries using Views. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 389–398, San Diego, California, USA, February 2000. IEEE Computer Society.

[36] Diego Calvanese, Giusepe De Giacomo, Mauricio Lenzerini, and Moshe Vardi. Query processing using views for regular path queries with inverse. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, Dallas, Texas, USA, May 2000. ACM Press.

[37] Diego Calvanese, Giusepe De Giacomo, and Maurizio Lenzeriniand Moshe Vardi. What is View Based Query Rewriting. In *Proceedings of the 7th International Workshop on Knowledge Representation meets Databases*, pages 17–27, Berlin, Germany, August 2000.

[38] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Answering queries using views in description logics. In *Knowledge Representation Meets Databases*, pages 6–10, 1999.

[39] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Description logic framework for information integration. To appear in Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98).

[40] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. What is view-based query rewriting? In Mokrane Bouzeghoub, Matthias Klusch, Werner Nutt, and Ulrike Sattler, editors, *Proceedings of the 7th International Workshop on Knowledge Representation meets Databases (KRDB 2000), Berlin, Germany, August 21, 2000*, number 29 in CEUR Workshop Proceedings, pages 17–27, 2000.

[41] Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA'93*, Washington, September 1993.

[42] R.G. Cattel. *The Object Database Standard: ODMG-93*. Morgan Kauffman, 1993.

[43] Online Computer Library Center. Dewey decimal classification. Available at www.oclc.org/dewey/.

[44] 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.

[45] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB*, Dallas, USA, May 2000.

[46] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of IPSJ Conference*, Tokyo, Japan, October 1994. TSIMMIS project: http://www-db.stanford.edu/tsimmis.

[47] C. Y. Chee, Y. Arens, C. A. Knoblock, and C. N. Hsu. Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[48] Rada Chirkova and Michael Genesereth. Linearly bounded reformulations of conjunctive databases. In *Proceedings of DOOD*, pages 987–1001, 2000.

[49] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, Dallas, USA, May 2000.

[50] Vassilis Christophides. Community Webs (C-Webs): Technological Assessment and System Architecture. Available at http://www.ics.forth.gr/~christop/CWebArch.ps.gz, September 2001.

[51] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On Labeling Schemes for Community Web Portals. Submitted for Publication.

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

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

[54] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, Seattle, Washington, June 1998.

[55] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, Rome, Italy, September 2001.

[56] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting Aggregate queries using views. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, pages 155–156, Philadelphia, Pennsylvania, 1999. ACM Press.

[57] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, Seattle, Washington, May 1998.

[58] C. Collet, M. Huhns, and W. Shem. Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, pages 55–62, December 1991.

[59] D. Connolly, F. van Harmelen, I. Horrocks, and D. L. McGuiness. DAML+OIL Reference Description. http://www.w3.org/TR/daml+oil-reference-20011218, March 2001.

[60] World Wide Web Consortium. Extensible Markup Language (XML 1.0 (second edition)), October 2000. http://www.w3.org/TR/REC-xml.

[61] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *The VLDB Conference*, pages 341–350, 2001.

[62] C-Web Project. http://cweb.inria.fr.

[63] The Upper CYC Ontology. http://www.cyc.com.

[64] Shaul Dar, Michael Franklin, Bjorn Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, pages 330–341, 1996.

[65] Jean Davoigneau, Renaud Benoit-Cattin, Xavier de Massary, Bernard Gauthiez, and Catherine Manigand-Chaplain. *THESAURUS DE L'ARCHITECTURE*. Sous-Direction des Etudes, de la Documentation et de l'Inventaire, direction de l'Architecture et du Patrimoine, Ministere de la Culture et de la Communication, 2000.

[66] Ministère de la Culture. http://www.culture.fr.

[67] S. Decker, D. Brickley, J. Saarela, and J. Angele. A Query and Inference Service for RDF. In *Proceedings of the W3C Query Languages Workshop*, Cambridge, Massachusetts, 1998.

[68] S. Deen. Data Integration in Distributed Databases. *IEEE Transactions on Software Engineering*, 13(7):860–864, July 1987.

[69] 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.

[70] Alin Deutch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 431–442, Philadelphia, Pennsylvania, USA, June 1999.

[71] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium, August 1998. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.html.

[72] Paul F. Dietz. Maintaining order in a linked list. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127, San Francisco, California, USA, May 1982.

[73] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proc. of the Sixteen Annual ACM Symposium on Theory of Computing (STOC'87)*, pages 365–372, New York, NY, USA, May 1987.

[74] M. Doerr. Authority Services in Global Information Spaces: A requirements analysis and feasibility study. Technical Report TR-163, Institute of Computer Science, Foundation for Research and Technology, Heraklion, Crete, Greece, February 1996.

[75] M. Doerr and N. Crofts. Electronic organization on diverse data - the role of an object oriented reference model. In *Proceedings of 1998 CIDOC Conference*, Melbourne, Australia, October 1998.

[76] Martin Doerr and Irini Fundulaki. SIS-TMS: A Thesaurus Management System for Distributed Digital Collections. In *Proceedings of the Second European Conference on Research and Advanced Technologies for Digital Libraries (ECDL)*, Heraklion, Crete, Greece, September 1998.

[77] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. *Principles of Knowledge Representation and Reasoning*, chapter Reasoning in Description Logics, pages 193–238. Studies in Logic, Language and Information. CLSI Publication, 1996.

[78] Denise Draper, Alon Levy, and Daniel S. Weld. The Nimble System. In *Proceedings of the International Conference on Database Engineering*, 2001.

[79] Dublin Core. http://www.nlm.nih.gov/research/umls.

[80] Oliver M. Duschka and Michael R. Genesereth. Answering Recursive queries using Views. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, Tuscon, Arizona, USA, 1997.

[81] Oliver M. Duschka and Michael R. Genesereth. Query Planning in Infomaster. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 109–111, San Jose, CA, USA, 1997.

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

[83] M. Evett, W. Andersen, and J. Hendler. *Parallel Processing for Artificial Intelligence*, chapter Providing Computational Effective Knowledge Representation via Massive Parallelism. Elsevier Science, 1993.

[84] W. Fan, G. Kooper, and J. Simeon. A Unified Constraint Model for XML. In *Proc. WWW10*, Hong-Kong, China, May 2001.

[85] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a Nutshell. In R. Dieng, editor, *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW)*. Springer-Verlag, 2000.

[86] Dieter Fensel, Stefan Decker, Michael Erdmann, and Rudi Studer. Ontobroker: Or How to Enable Intelligent Access to the WWW. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling, and Management*, Banff, Canada, April 1998.

[87] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. Trading between relational and XML. In *Proc. of the Int. WWW Conf.*, 2000.

[88] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[89] Daniela Florescu, Alon Levy, Dan Suciu, and Khaled Yagoub. Optimization of run-time management of data intensive web sites. In *Proceedings VLDB*, 1999.

[90] D.J. Foskett. *Readings in Information Retrieval*, chapter Thesaurus. Morgan Kaufmann, 1997.

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

[92] Irini Fundulaki, Bernd Amann, Catriel Beeri, Michel Scholl, and Anne-Marie Vercoustre. STYX : Connecting the XML World to the World of Semantics. In *Int. Conf. on Extending Data Base Technology (EDBT)*, Prague, Czech Republic, March 2002. Demo Presentation.

[93] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[94] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[95] Georges Gardarin, Antoine Mensch, and Anthony Tomasic. An Introduction to the e-XML Data Integration Suite. In *Int. Conf. on Extending Data Base Technology (EDBT)*, LNCS, pages 297–306, Prague, Czech Republic, March 2002.

[96] Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Journal of Distributed Computing, Special Issue for the Twenty Years of Distributed Computing Research*, http://www.cs.technion.ac.il/ hagit/podc20/, 2003.

[97] 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.

[98] Charles F Goldfarb. *The SGML Handbook*. Oxford University Press, 1990. ISBN: 0-19-853737-1.

[99] Google web page. http://www.google.com.

[100] Gosta Grahne and Alberto Mendelzon. Tableau Techniques for querying information sources through global schemas. In *Proc. of the Int. Conf. on Data Transaction (ICDT)*, volume 1540 of *Lecture Notes in Computer Science*, pages 332–347, Jerusalem, Israel, 1999. Springer Verlag.

[101] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, 1993.

[102] Thomas. R. Gruber. A Translation Approach to Portable Ontology Specifications. Technical Report KSL 92-71, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1993.

[103] N. Guarino. Understanding, Building, and Using Ontologies. A commentary to "Using Explicit Ontologies in KBS Developemtn", by Heijst, Schreiber, and Wielinga.

[104] N. Guarino. Formal Ontology and Information Systems. In N Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems*, pages 3–15, Trento, Italy, June 1998.

[105] N. Guarino. *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*, chapter Semantic Matching : Formal Ontological Distinctions for Information Organization, Extraction, and Integration, pages 139–170. Springer Verlag, 1998.

[106] N. Guarino, C. Masolo, and G. Vetere. OntoSeek: Content-Based Access to the Web. *IEEE Intelligent Systems*, pages 70–79, May/June 1999.

[107] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of ICDE*, pages 208–219, 1997.

[108] R. Shabo H. Kaplan, T. Milo. A comparison of labeling schemes for ancestor queries. In *Proc of the thirteen Annual Symposium on Discrete Algorithms (SODA'02)*, pages –, San Francisco, California, USA, January 2002. ACM/SIAM.

[109] J. Heflin, J. Hendler, and S. Luke. Reading Between the Lines: Using SHOE to Discover Implicit Knowledge from the Web. In *In Proc. of AAAI Workshop on Artificial Intelligence and Information Integration*, 1998.

[110] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[111] H. Hwang and U. Dayal. View Definition and Generalization for Database Integration in a Multibase System. *IEEE Transactions On Software Engineering*, 10(6):628–645, November 1984.

[112] International Guidelines for Museum Object Information: The CIDOC Information Categories. http://www.cidoc.icom.org/guide/.

[113] International Council of Museum Documentation (ICOM/CIDOC). http://www.cidoc.icom.org.

[114] Documentation - Guidelines for the establishment and development of monolingual thesauri. International Organization for Standardization, 11 1986. Ref. No ISO 2788-1986.

[115] Documentation - Guidelines for the establishment and development of multilingual thesauri. International Organization for Standardization, 2 1985. Ref. No. ISO 5964-1985.

[116] The KA2 Ontology. http://www.nlm.nih.gov/research/umls.

[117] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML Data. Technical Report 8/99, University of Manheim, 1999. Available at http://pi3.informatik.uni-manheim.de.

[118] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th International Conference on World Wide Web*, Hawai, May 2002.

[119] V. Kashyap and A. Sheth. *Cooperative Information Systems, Trends and Directions*, chapter Semantic Heterogeneity in Global Information Systems: the Role of Metadata, Context and Ontologies. Academic Press, 1998.

[120] M. Kaul. View System: Integrating Heterogeneous Information Bases by Object Oriented Views. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 2–10, Los Angeles, February 1990.

[121] Knowledge Interchange Format (KIF). http://logic.stanford.edu/kif/kif.html.

[122] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 1995.

[123] W. Kim and F. Lochovsky, editors. *Object-Oriented Concepts, Databases and Applications.* Addyson Wesley, 1989.

[124] W. Eliot Kimber. HyTime and SGML: Understanding the HyTime HyQ Query Language, August 1993. Availiable via anonymous ftp at ftp.ifi.uio.no /pub/SGML/HyTime.

[125] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *Proc. of the AAAI Spring Symposium on Informatio n Gathering in Distributed Heterogeneous Environments*, Stanford, CA, March 1995.

[126] K. Knight and S. Luk. Building a Large-Scale Knowedge Base for Machine Translation. In *Proc. of the American Association of Artificial Intelligence AAAI-94*, Seattle, WA, 1994.

[127] P.G. Kolaitis, D.L. Martin, and M.N. Thakur. On the complexity of the containment problem for conjunctive queries. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, Seattle, Washington, June 1998. ACM Press.

[128] Andreas Krall, Jan Vitek, and Nigel Horspool. Near optimal hierarchical encoding of types. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object Oriented Programming (ECOOP'97)*, pages 128–145, Finland, 1997. Springer-Verlag.

[129] Sandrine Lafois. Implantation et comparaison de structures de données pour l'interrogation d'arborescences volumineuses de termes (thesaurus)". Mémoire de Stage, SIR PE-CNAM-TELECOM, École doctorale EDITE, October 2000.

[130] C. Lagoze. The Warwick Framework : A Container Architecture for Diverse Sets of Metadata. *D-Lib Magazine*, July/August 1996.

[131] Laks Lakshmanan, Fereidoon Sadri, and Iyer Subramanianananian. A Declarative Language for Querying and Restructuring the World Wide Web. In *Proceedings of Post-ICDE IEEE Workshop on Research Issues in Data Engineering (RIDE-NDS)*, 1996.

[132] J. Larson. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions On Software Engineering*, 15(4):449–463, April 1989.

[133] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical Report REC-rdf-syntax-19990202, W3C, February 1999. W3C Proposed Recommendation.

[134] D. B. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

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

[136] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman et al., editors, *Proceedings of the twenty-second international Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann Publishers, 1996.

[137] A. Levy and M. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In *Proc. of the European Conference on Artificial Intelligence (ECAI-96)*, 1996.

[138] A. Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information systems. *Journal of Intelligent Information Systems*, 1995. Special Issue on Networked Information Discovery and Retrieval.

[139] Alon Y. Levy. *Logic Based Artificial Intelligence*, chapter Logic-Based Techniques in Data Integration. Kluwer Publishers, 2000. forthcoming.

[140] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of PODS*, San Jose, California, US, May 1995.

[141] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proc. of 27th International Conference on Very Large Data Bases(VLDB'02)*, pages 361–370, Roma, Italy, September 2001. Morgan Kaufmann.

[142] Witold Litwin, Leo Mark, and Rick Rousopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, 1990.

[143] D. Lomet and J. Widom, editors. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, volume 18-(2), June 1995.

[144] B. Ludscher, Y. Papakonstantinou, and P. Velikhov. A Framework for Navigation-Driven Lazy Mediators. In *ACM Workshop on the Web and Databases (WebDB'99)*. http://www-rocq.inria.fr/ cluet/WEBDB/procwebdb99.html, Philadelphia, USA, 1999.

[145] R. M. MacGregor. Inside the LOOM Description Classifier. *SIGART Bulletin*, 2(3), 1991.

[146] Alexander Maedche and Stefen Staab. Ontologies: Representation, Engineering, Learning and Applications. Tutorial presented in the 1st International Semantic Web Conference, June 2002.

[147] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[148] M. Maloney and A. Malhotra. RDF Query Specification. In *Proceedings of the W3C Query Languages Workshop*, Cambridge, Massachusetts, 1998.

[149] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Rome, Italy, September 2001.

[150] I. Manolescu, D. Florescu, and D. Kossmann. Pushing XML Queries inside Relational Databases. Technical Report 4112, INRIA, January 2001.

[151] Ioana Manolescu. *Optimization techniques for querying heterogeneous distributed data sources*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, December 2001.

[152] M. Marchiori and J. Saarela. Query + Metadata + Logic = Metalog. In *Proceedings of the W3C Query Languages Workshop*, Cambridge, Massachusetts, 1998.

[153] Amèlie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Migne. Change-centric management of versions in an xml warehouse. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proc. of 27th International Conference on Very Large Data Bases (VLDB'02)*, Roma, Italy, September 2001. Morgan Kaufmann.

[154] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. In IEEE Computer Society Press, editor, *Proceedings First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 14–25, Brussels, June 1996.

[155] Alberto Mendelzon, George Mihaila, and Tova Milo. Querying the World Wide Web. In *Conference on Parallel and Distributed Information Systems*, 1996.

[156] MERIMEE Thesaurus. http://www.culture.fr/documentation/merimee/accueil.htm.

[157] Mesmuses project. http://aquarelle.inria.fr/mesmuses/index.html.

[158] R.S. Michalski. *Categories and Concepts, Theoretical Views and Inductive Data Analysis*, chapter Beyond Prototypes and Frames: The Two-Tiered Concept Representation. Academic Press, 1993.

[159] A. Michard, V. Christophides, M. Scholl, M. Stapleton, D. Sutcliffe, and A-M. Vercoustre. The Aquarelle Resource Discovery System. *Journal of Computer Networks and ISDN Systems*, 30(13):1185–1200, August 1998.

[160] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sebastien Ailleret, Bernd Amann, and Amelie Marian. Acquiring XML pages for a Webhouse. In *BDA*, 2000.

[161] G. A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[162] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to WordNet: An On-line Lexical Database. ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.pdf, 1993. White Paper.

[163] A. Motro and P. Buneman. Constructing Superviews. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, number 56–64, Ann Arborm, April 1981.

[164] Namespaces in XML. http://www.w3.org/TR/REC-xml-names.

[165] D. Nardi and R. J. Brachman. *Description Logic Handbook*, chapter An Introduction to Description Logics, pages 5–44. Cambridge University Press, 2002.

[166] Members of the Topic Maps.Org Authoring Group. XML Topic Maps (XTM) 1.0. http://www.topicmaps.org/xtm/1.0, August 2001.

[167] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM) in conjunction with EDBT*, LNCS, Prague, Checzh Republic, March 2002. Springer. http://www.pms.informatik.uni-muenchen.de/publikationen/ PMS-FB-2002-4.

[168] Open Directory. http://www.dmoz.org.

[169] C. Paice. A Thesaural Model for Information Retrieval. *Information Processing and Management*, 27(5):443–447, 1991.

[170] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of International Conference on Very Large Databases (VLDB)*, pages 413–424, Bombay, India, September 1996. Morgan Kaufmann.

[171] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Data Engineering*, New Orleans, February 1996. TSIMMIS project: http://www-db.stanford.edu/tsimmis.

[172] Y. Papakonstantinou and V. Vassalos. Query rewriting using semistructured views. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, Philadelphia, Pennsylvania, USA, 1999.

[173] Platform for Internet Content Selection. http://www.w3.org/PICS.

[174] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries using Views. In *Proceedings of the International Conference on Very Large DataBases*, Cairo, Egypt, September 2000.

[175] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries using Views. *VLDB Journal*, 2001.

[176] Stephane Radicevic. Interfaces pour l'élection de liens par intégration d'une ontologie et de thesaurus. Mémoire d'Ingénieur, Conservatoire National des Arts et Métiers, January 2000.

[177] Anand Rajaraman, Yehoshua Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, San Jose, California, May 1995. ACM Press.

[178] Thesaurus of the Royal Commision of the Historical Monuments of England (RCHME). http://www.rchme.gov.uk/thesaurus/thes_splash.htm.

[179] W3C Technology and Society Domain : Resource Description Framework (RDF). http://www.w3.org/RDF/.

[180] The ICS-FORTH RDFSuite Web Site. http://139.91.183.30:9090/RDF.

[181] Retsina Semantic Web Calendar Agent. http://www.daml.ri.cmu.edu/site/projects/RDFCalendar.

[182] C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML heterogeneous data sources. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS)*, 2001.

[183] Philippe Rigaux, Michel Scholl, and Agnes Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers, 2001.

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

[185] The RDF Site Summary. http://groups.yahoo.com/groups/rss-dev/files/schema.rdf.

[186] R. Daniel Jr. (eds.) S. DeRose, E. Maler. XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, September 2001. http://www.w3c.org/TR/xptr.

[187] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[188] J. Sharma. Oracle8ispatial: Experiences with extensible databases. An Oracle Technical White Paper, May 1999.

[189] A. Sheth. *Interoperating Geographic Information Systems*, chapter Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. Kluwer, 1999.

[190] A. Sheth and V. Kashyap. So far (Schematically), yet So Near (Semantically). In *Proceedings of the IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems, DS-5*, IFIP Transactions A-25, Holland, November 1992.

[191] A. P. Sheth and J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[192] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, July 2001.

[193] M. Sintichakis and P. Constantopoulos. A Method for Monolingual Thesauri Merging. In *Proc. 20th International Conference on Research and Development in Information Retrieval, ACM SIGIR*, Philadeplphia PA, USA, July 1997.

[194] D. Soergel. The Art and Architecture Thesaurus, AAT. A critical appraisal. Technical report, College of Library and Information Sciences, University of Maryland, 1995.

[195] D. Srivastava, S. Dar, H.V. Jagadish, and A. Levy. Answering Queries with Aggregation using Views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 318–329, Bombay, India, September 1996.

[196] K. Stoffel, M. Taylor, and J. Hendler. Efficient Management of Very Large Ontologies. In *Proceedings of American Association for Artificial Intelligence Conference*, 1997.

[197] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, Madison, Wisconsin, USA, June 2002.

[198] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1.

[199] Robin Thornes. *Protecting Cultural Objects in the Global Information Society: The Making of Object ID*. Getty Information Institute, 1997.

[200] A. Tomasic, R. Amouroux, P. Bonnet, O. Kapitskaia, H. Naacke, and L. Raschid. The Distributed Information Search Component (DISCO) and the World Wide Web. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 546–548, Tuscon, Arizona, USA, June 1997.

[201] Antony Tomasic. XML/DBC: A Standard API for Access to XML Repositories and Mediators. Second International Workshop on Data Integration over the Web (DIWEB), May 2002. Invited panel presentation.

[202] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William Cody, Ron Fagin, Peter M. Schwarz, John Thomas, and Edward L. Wimmers. The Garlic project. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, volume 25, 2 of *ACM SIGMOD Record*, pages 557–558, New York, June 4–6 1996. ACM Press.

[203] United List of Artist Names. http://www.getty.edu/research/tools/vocabulary/ulan.

[204] Unified Modeling Language System. http://www.nlm.nih.gov/research/umls.

[205] IEEE Standard Upper Ontology. http://suo.ieee.org.

[206] Mike Uschold and Michael Gruninger. Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[207] MARC STANDARDS. http://lcweb.loc.gov/marc/marc.html.

[208] Pierangelo Veltri. *A View Mechanism for Large Scale XML Repositories: Design and Implementation.* PhD thesis, Université Paris XI, Orsay, October 2002.

[209] Vertigo database Group, CNAM-Paris. http://cedri.cnam.fr/vertigo.

[210] V. Vianu. A web Odyssey: from Codd to XML. In *Proc. of the Int. Conf. on Principle of Database Systems (PODS)*, Santa Barbara, CA, USA, 2001.

[211] W3C XML Query Working Group. W3C XML Query Requirements. Technical report, W3C, January 2000. http://www.w3.org/TR/2000/WD-xmlquery-req-20000121.

[212] S. Weibel, J. Miller, and R. Daniel. Dublin Core. OCLC/NCSA Metadata Workshop Report, 1995.

[213] World Wide Web Consortium. Document Object Model (DOM) 2.0. `http://www.w3.org/DOM/DOMTR`.

[214] World Wide Web Consortium. HyperText Markup Language (HTML) 4.1. `http://www.w3.org/MarkUp/`.

[215] World Wide Web Consortium. The W3C web site. `http://www.w3.org/`.

[216] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computers*, 25(3):38–49, March 1992.

[217] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[218] Misha Wolf, Ken Whistler, Charles Wicksteed, Mark Davis, and Asmus Freytag. A Standard Compression Scheme for Unicode. Technical Report Unicode Technical Standard 6, UNICODE, August 2002.

[219] James Wood. *Readings in Knowledge Representation*, chapter What's in a link? Morgan Kaufmann, 1985.

[220] Wordnet. http://www.cogsi.princeton.edu/wn.

[221] World Wide Web Consortium. W3C Semantic Web Activity. http://www.w3.org/2001/semweb-fin/w3c.

[222] Lucie Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, 2001.
`http://osage.inria.fr/verso/PUBLI/`.

[223] Yahoo web page. http://www.yahoo.com.

[224] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings VLDB*, pages 136–145, Athens, Greece, 1997.

[225] Yannis Papakonstantinou and Hector Garcia-Molina and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the International Conference on Data Engineering (ICDE)*, March 1996.