Thèse d'Informatique

soutenue le 21 septembre 2004 par

Omar Benjelloun

pour obtenir le grade de

Docteur de l'Université Paris XI

Active XML : Une approche des services Web centrée sur les données

(Active XML : A data-centric perspective on Web services)

Directeurs de thèse : **Serge ABITEBOUL & Tova MILO**

JURY

M.	Serge ABITEBOUL	Directeur de thèse	INRIA Futurs
M.	Catriel BEERI	Rapporteur	Hebrew U. of Jerusalem
Mme.	Véronique BENZAKEN	Examinateur	U. Paris XI
M.	Georges GARDARIN	Président du jury	U. de Versailles Saint-Quentin
M.	Alon HALEVY	Rapporteur	U. of Washington
Mme.	Tova MILO	Directeur de thèse	INRIA Futurs et U. of Tel-Aviv

Résumé

Cette thèse introduit Active XML (AXML, en abrégé), un modèle déclaratif mettant à profit la technologie des services Web pour la gestion de données distribuées et mis en oeuvre dans une architecture pair-à-pair.

Un document AXML est un document XML pouvant contenir des appels de services Web, dont l'invocation enrichit le document. Un service AXML est un service Web échangeant des document AXML. Un "pair" AXML est un entrepôt de documents AXML, jouant d'une part le rôle de client, en invoquant les appels de services inclus dans ses documents, et d'autre part celui de serveur, en fournissant des services AXML pouvant être définis comme des requêtes ou mises-à-jour sur ses documents. L'approche AXML permet de combiner harmonieusement données extensionnelles (en XML) et intensionnelles (les appels de services). La capacité des pairs AXML à échanger de tels documents conduit à un paradigme puissant de gestion de données distribuées.

L'approche AXML génère des questions importantes également étudiées dans cette thèse. Nous nous intéressons d'abord au contrôle de l'échange de documents AXML, en proposant l'utilisation de schémas déclaratifs et en fournissant des algorithmes pour leur vérification statique. Puis, nous envisageons l'évaluation "paresseuse" de requêtes sur des documents AXML, en introduisant des techniques analysant la pertinence pour les requêtes des appels de services inclus dans les documents.

Notre implantation des pairs AXML, conforme aux standards du W3C est également présentée dans la thèse.

Remarque. Cette thèse est rédigée en anglais. Une traduction en français de l'introduction et de la conclusion se trouve en Annexes B et C.

Mots-clefs

XML, Active XML, services Web, gestion de données, pair-à-pair.

Abstract

This thesis introduces Active XML (AXML, for short), a declarative framework that harnesses Web services for distributed data management, and is put to work in a peer-to-peer architecture.

An AXML document is an XML document that may contain embedded calls to Web services, whose invocation enriches the document. An AXML service is a Web service that exchanges AXML documents. An AXML "peer" is a repository of AXML documents. On the one hand, it acts as a client, by invoking the service calls embedded in its documents. On the other hand, a peer acts as a server, by providing AXML services that can be declaratively specified as queries or updates over the AXML documents of its repository. The AXML approach allows for gracefully combining stored information with data defined in an intensional manner (as service calls). The fact that AXML peers can exchange a mix of materialized and intensional data (via AXML documents) leads to a very powerful distributed data management paradigm.

The AXML approach leads to a number of important problems that are studied in the thesis. First, we address the issue of controlling the exchange of AXML data. We propose to use declarative schema specifications, and provide algorithms to statically enforce them. Second, we propose techniques for the "lazy evaluation" of queries on AXML documents, that detect which embedded service calls may contribute to query answers.

An implementation of AXML peers compliant with W3C standards is also described in the thesis.

Keywords

XML, Active XML, Web services, data management, peer-to-peer.

Merci,

Serge, pour m'avoir accueilli dans ton équipe et de m'avoir tant appris. Tu resteras mon modèle de générosité, d'intelligence et de discernement.

Tova, you've been my teacher, friend, and mother at the same time. Your presence brought joy and happiness to the three years we spent together.

Catriel and Alon, thanks for reading my manuscript and for your useful comments. I never thought people I admire so much would be my referees.

Georges, pour avoir présidé mon jury et Véronique pour en avoir fait partie. J'en suis très honoré et heureux.

A ceux qui ont guidé mes premiers pas vers le Fabuleux Monde de la Recherche : Alain Michard, grâce à qui tout à commencé. Michel, Vassilis, Irini et Bernd : Les premiers spécimens de chercheurs que j'ai observé et qui m'ont donné envie de devenir un drôle d'oiseau moi aussi.

A mes "grands frères" : Piero, Irini, Laurent, Vincent, Gregory, Benjamin. Vous m'avez donné l'exemple et l'amitié.

A mes "petits frères" : Vikas, Fred, Shobhit, Bogdan C., Emanuel, Bogdan M. Vous m'avez appris le plaisir de la transmission. Je vous souhaite une belle réussite.

A tous ceux qui participent au projet Active XML (c.f. http://activexml.net). Vos idées et vos efforts font sa force. Jérôme, merci pour ton aide précieuse et ton dévouement.

A tous les versos et les gemos, vous êtes vraiment une équipe formidable.

Dalila, merci de m'avoir donné la force.

Contents

List of Figures

Chapter 1

Introduction

One of the most essential issues in computer science is the management of data. Since the 60's, the database community has developed the necessary science and technology to manage data in central repositories. From the early days, many efforts have been devoted to extending these techniques to the management of distributed data as well, and in particular to its integration, e.g. [48, 103, 94].

But the Web revolution is setting new standards for data management. Managing data at the scale of the Web is dramatically changing the context, mainly because of (i) the high heterogeneity and important interoperability problems between sources and (ii) the autonomy of sources and the very large scale of the Web. *The goal of this thesis is to propose a new framework for data and service integration, that addresses these two concerns, and is therefore suitable for data management at the scale of the Web.*

Heterogeneity XML [109], as a *standard* data model and representation format for semi-structured data, raised a considerable interest among the data management community (e.g., [40, 67, 82]) for the flexibility it brings to solving heterogeneity issues. However, on the Web, significant interoperability problems come from the fact (i) that data sources often expose their data as a set of Web pages, mixing structure and presentation, and (ii) that this data is sometimes only accessible through forms. Promising new means of communication are emerging, that capture the publication of such data and the access to it, namely *Web services*. Web services are programs that accept XML parameters and return XML results. The SOAP [92] and WSDL [104] standards enable calling and describing these remote programs seamlessly over the Internet. *In our framework, XML and Web services are effectively leveraged for data management tasks.*

Autonomy Centralized architectures for data integration somehow contradict the essence of the Web, which is based on "loose-coupling", that is, distribution of control and independence of participating systems. Furthermore, centralized architecture hardly scale up to the large size of the Web. *Peer-based architectures*, in which resources are shared by direct exchange between systems, propose a credible alternative to centralized architectures, and are already spreading, notably in the file-sharing context [78, 45, 42]. These architectures capture the autonomous nature of systems, and their ability to act both as producers of information (i.e. as servers) and as consumers of information produced by others (i.e. as clients). *In our framework, peer-based architectures form the basis for scalable data management.*

We propose Active XML (AXML in short), a language that leverages Web services for data management and is put to work in a peer-to-peer architecture. The language introduces *AXML documents*, that are XML documents with embedded calls to Web services. Such documents are enriched by the results of invocations of the service calls they contain. The AXML model also defines *AXML services*; these are Web services that exchange AXML documents. Such services lead to powerful data-oriented schemes for distributed computation, where several systems can dynamically collaborate to perform a data management task, and in particular are able to discover new relevant sources of information at run-time.

After an overview of prerequisite material in Chapter 2, the AXML model will be presented and motivated in Chapter 3, where related works will also be considered. One should note that data with embedded calls to operations is not a new idea, and has even already been considered in the context of XML and Web services. For instance, in Microsoft Office XP, SmartTags within Office documents can be linked to Microsoft's *.NET* platform for Web services [87]. However, to our knowledge, the work presented in this thesis is the first proposal that actually turns calls to Web services embedded in XML documents into a powerful tool for distributed data management. By seamlessly combining *extensional data* (expressed in XML) and *intensional data* (the service calls, that provide means to get data) our approach notably captures different styles of data integration, such as warehousing, mediation and allows for flexible combinations of both.

Some important and novel issues arise when processing AXML documents and services, that need to be addressed in order to take advantage of the flexibility of this model:

- Intensional data is valuable. Therefore, systems that exchange AXML documents should be able to precisely control which parts of the intensional data should be made extensional before an exchange occurs. This choice may be influenced by various parameters, such as performance and security. We address this issue using a schema-based approach. A suitable extension of XML Schema [110] is defined, to specify the nature of the exchanged data. Automata-based algorithms are introduced, to make the exchanged documents conform to the specified schema. This part of our work will be the topic of Chapter 4.

- When queries are evaluated on AXML documents, a major challenge is to detect which service calls may bring data that is relevant to the query answer. The problem is intricate, as service calls may be embedded anywhere in the document, and service invocations possibly return data containing news calls. We propose algorithms to address this problem, that are based on rewriting the initial query. This "lazy" evaluation of queries on AXML documents will be presented in Chapter 5 of this thesis.

AXML documents and services nicely fit in a peer-to-peer architecture, where each system may act both as a client, by invoking the service calls embedded in its AXML documents, and as a server, by providing services over these documents. Building on this paradigm, we implemented a system (called the AXML peer) that is devoted to the management of AXML documents and services. The functionalities of AXML peers, both as clients and servers, as well as our implementation will be the topics of Chapter 6. Applications that have been considered to validate this approach will also be overviewed.

In the last chapter, we will conclude by placing our work into perspective, and mentioning some important issues that still need to be addressed.

Chapter 2

Preliminaries

We start with a brief presentation of the technologies that our work is based upon. Most of them are now covered by standards of the World Wide Web Consortium (W3C) [100], and commonly used in modern applications. Readers already familiar with these technologies can skip this section.

As hinted by its name, Active XML relies on *XML*, the eXtensible Markup Language, today's de facto standard for data exchange between applications. XML comes along with typing languages, such as *DTD*s and *XML Schema*, and also with query languages: *XPath* and *XQuery*. *Web services* is a term that denotes a family of XML-based protocols/languages to streamline the communication between distant machines/programs, by unifying the way they are invoked, described and discovered.

2.1 The eXtensible Markup Language (XML)

The starting point of the present work is the semistructured data model [56] and its current standard incarnation, namely XML [109]. XML is a model and a textual representation for semi-structured data, that was initially derived from SGML, the Standard Generalized Markup Language [58]. XML became a W3C standard in 1998, and is now used in a broad range of applications, either as an internal data representation model, or for communication

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss">

  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</title>
      <rss:pubDate>Feb 18, 2003</pubDate>
      <rss:description>
        Google just acquired Pyra labs,
        the company that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 2.1: A sample XML document

purposes, to exchange data between applications. Figure 2.1 is a sample XML document:

An XML document is a single-rooted ordered tree, consisting of *elements* (e.g. `newspaper`, `date`, `title`), *attributes* (e.g. `xmlns`, `id`) and *textual data* (e.g. `"Google goes Blog crazy"`). Elements, which form the structure of the document, are represented as tags that can have an arbitrary label. Each element has an ordered list of children, consisting of elements, textual data, or a mixture of both. Elements may also have an unordered set of attributes, whose values are necessarily textual data (e.g. `id="cx_ah_0_218"`).

Since XML documents can essentially be viewed as node-labeled trees, we will often represent them as such. Figure 2.2 shows a tree representation of our sample document (ignoring attributes).

XML is often compared to HTML, the Hyper-Text Markup Language [55], another (earlier) successor of SGML, which is the *lingua franca* of the Web. The main difference between the two languages is that HTML imposes

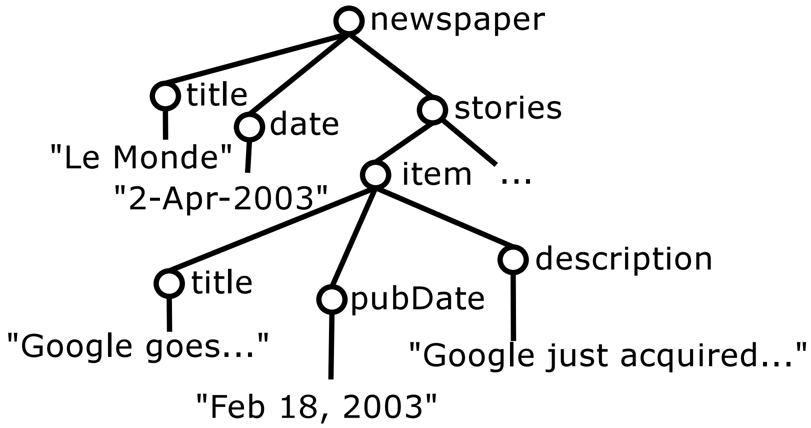

Figure 2.2: Simplified tree representation of an XML document

the use of a *fixed* set of element labels and attribute names, whose purpose is to allow the description of a structured hypertext document [1], while XML allows arbitrary labels to be used, and is therefore able to represent any tree-structured data. In other words (and retrospectively), HTML can be seen as a specialized "subset" of XML, whose purpose is to define hypertext documents. This is indeed how XHTML, the most recent incarnation of HTML, is defined [107].

2.1.1 Namespaces

To distinguish between multiple vocabularies, XML relies on a mechanism of *namespaces* [77]. Each namespace is uniquely identified by a URL. The default namespace in the document of Figure 2.2 is `http://www.lemonde.fr`, and applies to all elements in the document, unless otherwise specified.

Here, the `item` element and its descendants belong to the namespace `http://purl.org/rss`. References to a namespace previously declared in the document can be made using a *prefix* mechanism (here, `rss` is a prefix that refers to the `http://purl.org/rss` namespace). The namespace mech-

[1] Document must be understood here in the traditional sense (as in LaTeX for instance), i.e as a structured text devised for humans, containing sections, paragraphs, etc. A certain ambiguity comes from the fact that we also use the term document (in a broader sense) for XML data.

anism allows to unambiguously use labels that belong to several vocabularies. For instance, in our document, the use of different namespaces distinguish between the title of the newspaper, and the titles of the individual story items. In the presence of namespaces, the "full" name of an element, which is often called its *qualified name* (or QName) is the concatenation of the namespace it belongs to and of its label.

2.2 Typing XML

XML is indeed a very flexible data representation language, as it allows one to represent arbitrary trees. However, to ensure that XML data is correctly processed by a given program, or conforms to some exchange specification, one often needs to control the structure of the processed/exchanged documents, i.e. which elements/attributes/data values are allowed to appear in a document, and in what position. For instance, to define XHTML, the XML version of HTML, it is necessary to restrict the possible element labels to those defined by HTML (e.g. `<html>`, `<table>`, `<h1>`, etc.), and to enforce the constraints of the HTML syntax (e.g. that list elements `<ol>` or `<ul>` have only children that are list item elements `<li>`). There has been a number of proposals for typing XML documents, e.g., [66, 54, 34]. In the Active XML context, we selected XML Schema [110] for several reasons. First, it is the standard recommended by the W3C for describing the structure of XML documents. Furthermore, it is the typing language used (in WSDL) to define the signatures of Web services [104]. We will first present DTDs, a simpler typing language that we will use in Chapters 4 and 5 to illustrate our algorithms, then move to XML Schema, which is actually used in our implementation.

2.2.1 Document Type Definitions (DTDs)

Inherited from SGML, DTDs were historically the first schema language proposed and widely used for XML. Their main advantage is their simplicity, which makes them easy to learn and use. In a nutshell, a DTD (i) defines the set of element labels allowed to appear in a document, and (ii) describes the children allowed for each element of a given label, by a regular expression over element labels (plus the keyword PCDATA for textual data). The document of Figure 2.1 (ignoring the namespaces) is valid with respect to the DTD of Figure 2.3.

```
<!DOCTYPE newspaper [
     <!ELEMENT newspaper (title,date?,stories)>
     <!ELEMENT title (#PCDATA)>
     <!ELEMENT date (#PCDATA)>
     <!ELEMENT stories (item*)>
     <!ELEMENT item (title,pubDate?,description?)>
     <!ATTLIST item id ID #required>
     <!ELEMENT pubDate (#PCDATA)>
     <!ELEMENT description (#PCDATA)>
     ]>
```

Figure 2.3: A sample Document Type Definition (DTD)

The main drawback of DTDs is their limited expressive power. In a DTD, each element label is associated with a single type, and therefore cannot have a different structure (e.g., based on its context in the document). Also, textual data is not typed: it is not possible to say, for instance, that the price of a car must be a positive integer. Also, they do not account for namespaces.

2.2.2 XML Schema

DTDs are progressively being replaced by XML Schema, the current W3C standard for XML type descriptions. XML Schemas improve on DTDs by providing a stronger expressive power (mainly through specialization, and atomic types), and a better modularity (through the use of namespaces, and a schema import mechanism). Also, XML Schemas are expressed using an XML syntax. Their main weakness is certainly their richness and complexity, and sometimes rather unclear semantics (e.g. the 1-unambiguity rule says that no element should have two children with same label but with different types). Figure 2.4 gives a sample XML Schema. The document of Figure 2.1 is valid with respect to this schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.lemonde.fr"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:rss="http://purl.org/rss"
           elementFormDefault="unqualified"
           attributeFormDefault="unqualified">

<xs:import namespace="http://purl.org/rss"
           location="http://purl.org/rss/rss-schema.xsd"/>

<xs:element name="newspaper">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="date" type="xs:date" />
      <xs:element name="stories" type="ItemList"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="ItemList">
  <xs:sequence>
   <xs:element ref="rss:item" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

Figure 2.4: A sample A sample XML Schema

This schema illustrates a number of features of the XML Schema language:

- The use of namespaces, to precisely reference elements from external schemata, and a schema import mechanism, used to actually retrieve the definitions of externally defined namespaces (e.g. the `rss:item` element is defined externally, in a schema that can be retrieved from the `http://purl.org/rss/rss-schema.xsd` location).

- Rich atomic types, by contrast with the single `PCDATA` atomic type of DTDs. For instance, our sample schema ensures that the content of the `date` element is actually a valid date.

- Precise cardinalities, defined by the `minOccurs` and `maxOccurs` attributes.

2.3 Querying XML

XML data can of course be manipulated from programs. Standard programmatic interfaces exist for that, viewing a document either as a stream of events encountered while sequentially reading a document (see SAX, the Simple API for XML [90]) or as a tree of complex objects (see DOM, the Document Object Model [39]). However, as for relational data, users (and application developers) are more comfortable and productive with higher level, declarative *query* languages. Disparate efforts to define a query language for XML are now unifying in the XQuery proposal [112], and its subset XPath [111].

2.3.1 XPath

XPath [111] is probably the most used query language for XML, either by itself or embedded in more complex languages, such as XQuery [112] or XSLT [113]. The core of XPath allows to express variable-free regular path queries [8], i.e. regular expressions over the sequences of labels obtained by traversing a document along a number of axes: parent/child, ancestor/descendant, previous/next-sibling, etc. XPath also allows the use of existential predicates, and (in)equality tests on atomic values. Figure 2.5 shows a sample XPath query, in its commonly-used abbreviated syntax [2].

[2] The full syntax gives explicitly the names of the followed axes (e.g. descendant, next-sibling) and allows to express more queries than the abbreviated syntax used here. See [111] for details.

```
/newspaper//item[pubDate ="Feb 18, 2003"]/title
```

Figure 2.5: A sample XPath query

This query asks for the `titles` of all `items` that are (i) descendants of the root `newspaper` element, and (ii) that were published on February 18^{th}, 2003. Such a query (as well as a large sub-set of XPath queries) can be represented as a *tree pattern*, as shown in Figure 2.6. Regular edges denote a parent-child relationship, while doubled lines represent descendant edges, and the output node of the query is pointed by an arrow. The semantics of the evaluation of such an XPath query on a document is generally defined as the set of all possible *embeddings* of the tree pattern query into the tree of the document, projected on the output node (See, for instance, [69] for a formal definition).

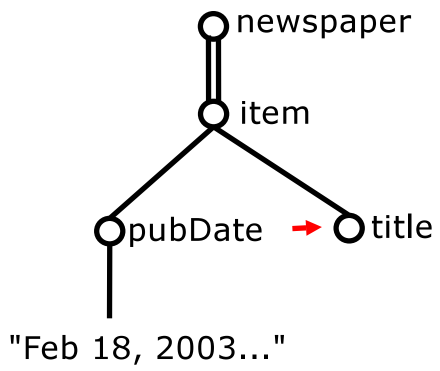

Figure 2.6: Tree representation of an XPath query

2.3.2 XQuery

XQuery [112] is a full-fledged query language for XML, meant to become the equivalent of SQL for XML. It builds on the path selection features of XPath, and adds to that the ability to use multiple variables, to express conditions involving the variables, and to construct complex results. XQuery is very rich and has many features (e.g. it allows for nested queries), in order to accommodate a very broad spectrum of queries.

```
for $a in document("newspaper.xml")/newspaper//item,
    $b in $a/pubDate,
where $b >= "Feb 1st, 2003"
return <digest>
          <story>{$a/title, ": ", $a/description}</story>
          </digest>
```

Figure 2.7: A sample XQuery query

Figure 2.7 shows a sample XQuery query. The `for` part selects absolute or relative XPath expressions into variables (denoted with a $ sign). The `where` part defines selection conditions on these variables. The `return` part constructs the XML result of the query. Note that XPath expressions are not restricted to the `for` clause. They can also appear in the `where` and `return` clauses.

This query looks for all news items that were published after Feb. $1^{\{st}$, 2003, and outputs a digest of them, consisting of the concatenation of their title and description, placed below a story element.

2.3.3 X-OQL

X-OQL [12, 13] is a query language for XML that was designed and implemented in the Gemo team. We mention it here essentially because, for historical reasons, it is the main language used for queries in our implementation. We used X-OQL essentially because, at the time we started, no XQuery processor was available, and indeed the language was not yet completely defined. X-OQL was inspired by OQL, the standard query language for object databases [31], and by Lorel [11], one of the first query languages for semi-structured data. Its syntax is very similar to that of SQL, with a `select...from...where` block as main construct. The previous query is expressed in X-OQL as:

One can see that the structure is very similar to that of an XQuery query. The main visible differences are the following:

```
select <digest>
          <story>{$a/title, ": ", $a/description}</>
       </>
from $a in document("newspaper.xml")/newspaper//item,
     $b in $a/pubDate,
where $b >= "Feb 1st, 2003"
```

Figure 2.8: A sample X-OQL query

- The construction of the result is done in the `select` block, while the assignments of paths to variables is done in the `from` block.
- Labels of closing tags are omitted in the construction part of the result
- Variables need not be prefixed with a $ sign, although we often use such a prefixing for readability of the query.

In terms of features, X-OQL also supports nested queries, and is extensible through external functions. The implementation we use works in memory, and operates on a DOM representation of the queried document. We refer the reader to [12, 13] for a detailed description of the X-OQL language and its implementation.

2.4 Web Services

The Web has become a planet-scale distributed information system easy to use for humans. The vision of Web services is to achieve the same objective for machines. The idea is to build on the principles of the Web (simplicity, loose coupling, standardization, communication based on textual messages) to create an infrastructure for large-scale distributed computing between heterogeneous systems. The main target of such a *service-oriented architecture* is of course E-Commerce, and mainly business-to-business (B2B) interactions. The ultimate objective is to have a world-wide marketplace where, for instance, companies would dynamically find the suppliers that best fit their needs and are most competitive, and establish a business relationship with them, using as little human intervention as possible. While we are not directly concerned with E-Commerce specific aspects in this work, this approach has a side-effect that we view as crucial: it standardizes the way heterogeneous, remote sources of information (e.g. databases) can be accessed.

Various big industry players proposed infrastructures to support this Web services vision, e.g. .NET by Microsoft, e-speak by HP, SunOne by Sun Microsystems. They are now converging towards using a small set of core specifications to achieve interoperability [3]. Among those, we are directly concerned with the Simple Object Access Protocol (SOAP) [92] and the Web Services Description Language (WSDL) [104], that are used in our implementation, and will be presented next. Other aspects related to Web services, like their discovery, composition and semantics can be leveraged in the Active XML approach. E.g., UDDI registries [97] can be used by our system to publish or discover Web services of interest. They will be briefly overviewed as part of the related works in Chapter 3.

2.4.1 SOAP

SOAP [92] is an XML-based protocol for the exchange of messages between applications. It is stateless, unidirectional and independent of the particular transport protocol used for communication. It was designed to be independent of any particular computing platform (e.g. Unix, Microsoft Windows, Apple Macintosh), to take advantage of the protocols of the Web (e.g. HTTP, SMTP), and to be extensible. SOAP is soon to become a W3C recommendation.

SOAP is based on an XML syntax for messages exchanged between applications. The specification defines a precise XML syntax (associated to a particular namespace) for the *envelope* of the message, and lets the body of the message be arbitrary, application-specific XML data. A SOAP message consists of:

[3] An organization was even created for that. See http://www.ws-i.org.

- a *SOAP Header*: holding metadata for one or more recipients of the message and attributes to specify who must process these metadata.
- a *SOAP Body*: the application specific part of the message
- embedded in the SOAP body, possibly one or more *SOAP faults*, to report errors that may have occurred during the processing.

SOAP also specifies default means to encode in XML common data structures of programming languages, like arrays, records, etc. This encoding eases the automatic serialization (resp. deserialization) of such data structures into XML (resp. from XML) by the SOAP implementations that exist for various languages, and helps for interoperability across languages, e.g. so that a C array can be transparently used as a Java array when sent through a SOAP message.

The SOAP specification also defines bindings for common transport protocols, both synchronous ones such as HTTP, or asynchronous ones, such as SMTP or JMS. These bindings specify a number of conventions on how SOAP messages are exchanging information using each specific protocol. For instance, the HTTP binding specifies that SOAP headers must be sent as part of the HTTP headers, and that for RPC-style invocations, the HTTP-POST method should be used to send a message containing the request, which should be answered synchronously with an HTTP-OK answer containing the response.

Figure 2.9 shows such a SOAP RPC request over HTTP, that asks for the stock price of the IBM share.

The SOAP binding for HTTP specifies that in the case of RPC invocations, the first element of the body (here, `m:GetStockPrice`) is the qualified name of the invoked method, and that the elements below it are the parameters of the invocation.

Figure 2.10 shows the response to this request. SOAP specifies that the first element should have the same name as the invoked method, with the string "Response" appended to it, and should contain the answer to the invocation.

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap; charset=utf-8

<?xml version="1.0">
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
       soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figure 2.9: A SOAP RPC Request over HTTP

```
HTTP/1.1 200 OK
Connection: close
Content-Type: application/soap; charset=utf-8
Date: Mon, 28 Sep 2002 10:05:04 GMT

<?xml version="1.0">
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-encoding"
        soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:price>
    </m:GetStockPriceResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2.10: The SOAP Response to this request

SOAP offers a number of other features, like message exchange patterns (MEPs), that provide for the specification of interactions involving the exchange of several messages between multiple participants, using finite state machines. But these are of secondary interest in the context of Active XML. We will mainly focus on RPC-style synchronous invocations over HTTP, and indeed this is how most services available today are provided.

We mention however that, as we will see in Chapter 6, Active XML support and provides asynchronous (and even continuous) Web services, on top of the synchronous HTTP layer. Therefore, our framework can naturally be extended to support asynchronous SOAP services using real asynchronous transports, such as SMTP or JMS, as soon as they become widely available.

2.4.2 WSDL

The Web Services Description Language (WSDL) is the language used by Web services to describe the operations they provide. It is similar in spirit to the IDL language for CORBA. In short, WSDL provides all the necessary information to invoke a Web service, and describes the signatures of the operations it supports, in terms of the corresponding input and output messages, whose types are specified using XML Schema. The WSDL description of a service also specifies bindings to the particular protocols and transports used to access the operations of the service.

A WSDL specification is an XML document that consists of a number of sections:

- The *types* section specifies any specific schemas of messages that will be exchanged by the service using the XML Schema language.

- Each kind of *message* exchanged by the service is also described. Messages are composed of one or more *parts,* that have a specific XML Schema type.

- *Operation* elements define the operations supported by the service, and specifies their input and/or output messages. The presence and the order of these input and output messages determines the interaction pattern of the operation. For instance, an operation that corresponds to an RPC invocation (named “request-response” in the WSDL jargon) will have an input message, followed by an output message, while a notification operation will only have an output message.

- *Port types* are groups of operations that will be accessed through the same protocol.
- *Bindings*: associate a port type (i.e. a group of operations) with a particular protocol (e.g. SOAP) that will be used to provide the operations.
- *Ports*: associate an endpoint address to a binding, e.g. a URL if the port type is bound to a protocol that uses an HTTP transport, or an e-mail address in case SMTP is used.
- a *service* is a set of ports.

Figure 2.11 gives a sample WSDL description for the stock quote service.

The rather complex and hierarchical structure of such a description is mainly due to a quest for genericity and extensibility. The WSDL language tries to be as agnostic as possible regarding the particular protocol used to access the services (e.g. SOAP, or XML-RPC), the transport that will be used (e.g. HTTP or JMS) and even the schema language used to describe the types of messages e.g. XML Schema or RELAX [66]).

However, most of this WSDL code is generally not written manually, but generated automatically by the Web service middle-ware, from the code of the programs whose functionalities are exposed as Web services. As mentioned earlier, the most delicate part of this task (which is often done manually) is to pick XML Schema types for the messages, and to be able to do the corresponding serialization and deserialization tasks.

In Active XML, WSDL will play a crucial role to analyze the input and output types of services, in order to control the exchange of AXML data between applications, and to optimize the evaluation of queries on AXML documents, as will be seen in Chapters 4 and 5 respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://example.com/stockquote.xsd"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema targetNamespace="http://example.com/stockquote.xsd">
      <xsd:element name="TradePriceRequest">
        <xsd:complexType>
          <xsd:element name="tickerSymbol" type="string"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="TradePrice">
        <xsd:complexType>
          <xsd:element name="price" type="float"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd:TradePrice"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>
```

Figure 2.11: A sample WSDL description

We overviewed in this chapter the main technologies that our work is based upon, namely XML, and the associated typing and query languages, and core components of Web services. We are now ready to present and motivate the Active XML model. At the end of next chapter, once the reader is familiar with our language and its capabilities, we will mention some related works.

Chapter 3

The Active XML model

In this chapter, we introduce the fundamental components of the Active XML language, namely AXML documents and AXML services. We illustrate through examples their basic operation, and the flexibility they bring to distributed data management. We motivate the need for this new model, and compare it to related existing approaches for data management.

3.1 Active XML documents

Active XML documents are based on the simple idea of embedding calls to Web services inside XML documents. More precisely, we define an XML syntax to denote service calls, and allow elements conforming to this syntax to appear anywhere in a document. Intuitively, these calls represent some (XML) information that is not given extensionally, but *intensionally*, by providing means to get the corresponding data. Service calls can be *materialized*, which means that the associated Web service is invoked (using the SOAP protocol), and the results it returns enrich the document, at the location of the service call.

Figure 3.1 is a simple AXML document, that represents a local newspaper homepage. Note that this document is a syntactically valid XML document, where service calls are the elements labeled `call`. Service calls, represented in bold face, use the namespace `http://purl.org/net/axml`, to unambiguuously differentiate them from the rest of the data. We generally refer to this namespace using the prefix axml.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://purl.org/net/axml">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <weather>
    <axml:call service="forecast@weather.com">
      <city>Paris</city>
      <unit>Celsius</unit>
    </axml:call>
  </weather>
  <exhibits>
    <axml:call service="getEvents@TimeOut.com">
      exhibits
    </axml:call>
  </exhibits>

  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</title>
      <rss:pubDate>Feb 18, 2003 10:36:03 GMT</pubDate>
      <rss:description>Google just acquired Pyra labs, the company
        that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 3.1: An Active XML document

This document consists (i) of some extensional information: the title and date of the newspaper, and a news story about Google, and (ii) of intensional information: a service call to get the weather forecast, and another one to get a list of current exhibits from a local guide.

For each service call, the Web service that needs to be called is identified by the `svc` attribute of the `axml:call` element. For instance, `getEvents@TimeOut.com` means that we want to invoke the service `getEvents` provided by the `TimeOut.com` server. This is actually a simplified syntax, used here for the sake of clarity. The full syntax accounts for all the information needed to invoke the service using the SOAP protocol (as defined for RPC calls in Section 2.4.1). The subtrees below service calls are the *parameters* of the calls. For instance, the call to `forecast@weather.com` uses the name of the city `Paris` and a unit as parameters. In general, parameters can be arbitrary XML data, and we will see further on that they may even contain service calls themselves.

To illustrate the meaning of these service calls, we can further extend the analogy with HTML documents we used in Section 2.1. Just like hyperlinks use a special markup in HTML documents (<a href="http://...">), that browsers are able to interpret in order to display them as links, and provide enough information (i.e the URL present in the `href` attribute) to get their data when the user clicks on them, AXML service calls use a special unique markup, that can be recognized and processed by an AXML system. This provides enough information to actually invoke the Web service.

What is important to see at this point is that calls to *any* Web service can be inserted in AXML documents, thus taking advantage from the wealth of already existing services (such as Google, Amazon, e-bay, etc.), and leveraging them as useful data sources. Moreover, for data sources that are not already available as Web services, wrapping them as Web services has the virtue of making them more easily reusable for future integration tasks, in a format that is standardized and makes them interoperable with other systems.

3.1.1 Materialization

A service call embedded in an AXML document can be activated, by invoking the corresponding service, and the result of this invocation is used to enrich the document. This process is termed *materialization*.

Technically, the service invocation is done using the SOAP protocol, based on the attributes of the *call* element, that contain all the information needed to issue the SOAP call. The sent SOAP message also contains the values of the parameters (i.e the children of the *call* element). The service answers this request with a SOAP message, that contains the result of the invocation, or an error message. If no error occurred, the result is inserted in the document, *in place of* the service call element. Figure 3.2 shows our sample document after the materialization of the call to `forecast@weather.com`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://purl.org/net/axml">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <weather>
    <temp>16</temp>
  </weather>
  <exhibits>
    <axml:call service="getEvents@TimeOut.com">
      exhibits
    </axml:call>
  </exhibits>
  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</title>
      <rss:pubDate>Feb 18, 2003 10:36:03 GMT</pubDate>
      <rss:description>Google just acquired Pyra labs, the company
        that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 3.2: After invoking `forecast@weather.com`

We will see in Chapter 6 that other strategies of materialization are possible:

- The `call` element does not have to be *replaced* by the result of the invocation (e.g. it can be kept for future reuse).
- Invocations do not have to be synchronous, in the style of Remote Procedure Calls, but are allowed to be asynchronous, and can also be subscriptions (i.e. a single invocation results in a stream of several, asynchronous answers)

But for now, we assume that materialization simply consists in replacing a piece of intensional information (the service call), by its corresponding extent (the result of the synchronous invocation of the referenced Web service).

We now give a formal definition of AXML documents, which will be the one used in the rest of this thesis (unless otherwise notified) and illustrate the materialization process on this definition.

3.1.2 Formal definition

We model AXML documents as ordered labeled trees consisting of two types of nodes: data nodes and function nodes. The latter correspond to service calls. We assume the existence of some disjoint domains: $\mathcal{N}$ of nodes, $\mathcal{L}$ of labels, $\mathcal{F}$ of function names [1], and $\mathcal{D}$ of data values. In the sequel we use v, u, w to denote nodes, a, b, c to denote labels, and f, g, q to denote function names.

Definition 3.1.1 *An AXML document d is an expression (T, λ), where $T = (N, E, <)$ is an ordered tree. $N \subset \mathcal{N}$ is a finite set of nodes, $E \subset N \times N$ are the edges, $<$ associates with each node in N a total order on its children, and $\lambda : N \rightarrow \mathcal{L} \cup \mathcal{F} \cup \mathcal{D}$ is a labeling function for the nodes, where only leaf nodes may be assigned data values from $\mathcal{D}$.*

Nodes with a label in $\mathcal{L} \cup \mathcal{D}$ are called data nodes while those with a label in $\mathcal{F}$ are called function nodes. The children subtrees of a function node are

[1] As for the `svc` attribute used in our example, we assume in this model that function names identify Web service operations. This translates in the implementation to the SOAP parameters (URL, operation name, ...) that allow one to invoke the Web service.

the *function parameters*. Materialization is defined as previously: When the function is called (i.e. the service is invoked), the function parameters are passed to it. The return value then replaces the function node in the document. This is illustrated in Figure 3.3, where part of our sample document is represented as a tree. Data nodes are represented by circles, function nodes are represented by squares, and data values are quoted. Here, the *GetTemp* Web service is invoked with the city name as a parameter. It returns a *temp* element, which replaces the function node.

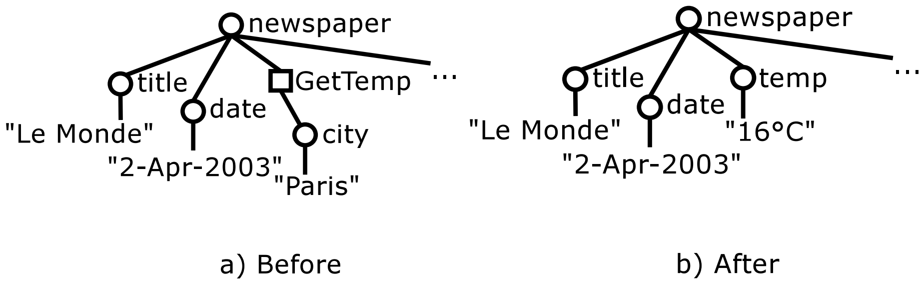

Figure 3.3: An AXML document before/after a call.

Note that we ignore in this model some features of XML such as attributes and namespaces. However, these features can be easily encoded in our simple model, which captures the richness and flexibility of (Active) XML. And indeed, our implementation uses and supports the full-fledged XML data model.

3.1.3 Motivations for AXML documents

We now highlight some important motivations for representing information as AXML documents. The key benefit of AXML data is the fact that some of the information is not represented extensionally but *intensionally*, i.e. by providing the means (in our case, the service calls) to obtain the corresponding extensional information. This enables a system/user that defines or receives AXML data to *reuse* autonomously some parts of the corresponding information, by giving her more knowledge about its origin or means to get it.

Dynamicity Perhaps the main motivation for representing information as AXML documents is to capture the fact that parts of the data may change with time. For instance, in our newspaper example, the temperature in Paris may change daily, and the news headlines may change every hour. This is captured by the fact that this part of the information is represented intensionally. The same service called at different times may yield different answers (e.g., because the external data source changed), so the same document at different times will have different semantics, thus reflecting world changes. In other words, whenever an AXML document is materialized, the service calls it contains will yeld up-to-date answers. An important question that arises in this context is when to activate each service call. We will provide means to control this in Chapter 6, where we will see how AXML systems manage *persistent* AXML data.

Knowledge Intensional information also conveys more knowledge than extensional information. Our newspaper AXML document can also be seen as a way to inform the system/user that she can obtain the temperature in Paris *at any time*. Moreover, she may learn something even beyond that: namely how to obtain the temperature *of any city*. In general, by obtaining the information in an intensional form, the client becomes more independent knowledge-wise and may be able to *generalize* the use of the service, by reusing it over time, or calling it with different parameters, e.g. to obtain news about other kinds of events from the `TimeOut` local guide (e.g. concerts, Movie hours, etc.).

We implicitly considered, in the previous examples, that systems/users could not only represent information locally as AXML documents, but that they could also *receive* such information from other systems. Indeed, the fact that systems can exchange intensional information is a central aspect of our approach. And since Web services are the natural way for systems to communicate in Active XML, we use them as the main vector for the exchange of intensional information. We next introduce AXML services, that are Web services able to exchange AXML documents.

3.2 Active XML services

As mentioned earlier, Active XML documents are syntactically valid XML documents, and therefore can be parsed and processed by any XML-aware system. Moreover, since Web services are invoked and described in a standard way, following the SOAP and WSDL specifications, then any Web services-aware system is able to materialize the calls embedded in AXML documents. Hence, AXML documents can be universally understood, and therefore can be exchanged between systems. This crucial property of AXML documents led us naturally to introduce AXML services.

AXML services are simply Web services that are able to exchange AXML documents. That is, they accept AXML documents as input parameters, and return AXML documents as results. We will show that this simple idea introduces a lot of flexibility in terms of interactions between the participating systems. We focus here on the benefits obtained from using such AXML services. The issue of implementing them will be addressed in Chapter 6, where convenient means to define them will be presented.

Note that, since AXML documents are (syntactically speaking) XML documents, AXML services are compliant with all the standards of Web services: SOAP messages containing service calls are still valid SOAP messages, and these services can still be described in WSDL. We will see in Chapter 4 an extension of XML Schema with intensional types, that can be used to describe the signatures of such AXML services in their WSDL description.

3.2.1 Recursive materialization

Let us first revisit the notion of materializing an AXML document, introduced in the previous section, in the presence of AXML services. Materialization now becomes a recursive process, since calling an AXML service may return some data that contains new service calls. The latter need to be materialized, and so on recursively. This is illustrated in Figure 3.4, which gives the state of our sample document after the invocation of the `getEvents@TimeOut.com` service. This service answered with a list of exhibits, plus a service call to get more exhibits from `Yahoo`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://purl.org/net/axml">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <weather>
    <temp>16</temp>
  </weather>
  <exhibits>
    <exhibit>
      <name>Naive Painting in Ancient Grece</name>
      <location>Le Louvre</location>
      <from>25-Apr-2003</from>
      <to>25-May-2003</to>
    </exhibit>
    ...
    <axml:call service="getExhibits@Yahoo.com">
      <city>Paris</city>
    </axml:call>
  </exhibits>
  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</title>
      <rss:pubDate>Feb 18, 2003 10:36:03 GMT</pubDate>
      <rss:description>Google just acquired Pyra labs, the company
        that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 3.4: After invoking `getEvents@TimeOut.com`

Since materialization of AXML documents becomes recursive, an important issue that arises is the one of termination: Can we "fully" materialize a document? Can we know (decide) in advance if the materialization of a document can be performed in a finite time? This termination problem is actually undecidable in the general case, when no assumptions are made on the properties of the Web services involved. We will see in Chapter 6 some restrictions on both the data model and the allowed services under which termination becomes decidable.

However, the fact that the termination of materialization is undecidable is not necessarily bad news. Non-termination is an inherent characteristic of most computations on the Web, due to its huge size and its ever-changing nature. Yet, the Web is a most useful (and studied) environment for computation! We now present some important motivations for AXML services.

3.2.2 Motivations for AXML services

AXML services are a key aspect of the AXML approach to distributed data management, because they enable the exchange of intensional information between participating systems. This exchange of AXML documents is the main novelty of our proposal, and we argue that it really brings a lot of of flexibility to distributed data management. We first consider the benefits obtained from having services return AXML answers, before turning to the use of AXML documents as input parameters.

3.2.2.1 AXML answers

All motivations we gave for AXML documents are of course important motivations for services returning AXML answers, since they they let their clients take advantage from the dynamicity and knowledge features inherent to AXML documents. We will now present some motivations that are specific to AXML services, and can be organized into two families: (i) the ability to represent partial answers and (ii) the possibility to enrich answers.

First, services that return AXML documents are able to provide their clients with a partial representation of their answer, instead of a fully computed one. This is interesting in many aspects:

Summarization If the client of the service is a human user, then the full answer may simply be too large and unreadable for her. The natural way to summarize it is to "intensionalize" some of it. We are here at the border with man machine interfaces. AXML may be viewed as an environment to provide several views of a service answer, as well as means to navigate between them, by zooming in and out. This can easily be accomplished using AXML, with the service calls acting as navigational links.

Incompleteness Another important reason is that we may want the answer to be incomplete. For instance, the answer may be very large and we may be satisfied with just some of it, e.g., we ask the query "XML" to Google and there are 22 300 000 answers. The server decides not to send them all but provide, say the first 10, and a link (i.e. a service call) to obtain more. This may be viewed as a very simplistic form of intensional data but the situation does not have to be so simple. Consider a search engine that is aggregating three different data sources. It receives a query and one of the sources is down. Then the search engine may want to answer in AXML with the concrete data it knows of, and with a call to the last source with the meaning that more data may be obtained from there when the source will come back up.

Partial computations AXML data can not only represent missing information, but also computations that still need to be carried out to obtain the full answer. Perhaps the best example of this is security and encryption. Suppose I ask for your secret phone number, you want to send it encrypted. So I need to decrypt your answer to understand it. This may be represented as a service call that has two parameters, holding(possibly intensionally) the data to decrypt on the one hand, and the secret key to decrypt it on the other hand. This service can only be evaluated by me, since I jaleously guard the secret key needed to perform the decryption.

As another example, some news systems provide as results some lists of news headlines, each of them with a summary and the means to buy the article (which again can be embodied by a service call). In AXML, these unaccomplished tasks may be viewed as part of the answer, to be obtained by the data receiver via some additional computation, that is, calling the service and paying the required fee.

We now turn to another family of motivations for services returning AXML answers, based on the idea of enriching the service answer with intensional parts that provide the client with extra functionality, in order to better fit her needs.

Metadata Consider, as an example of a "raw" service answer, the result of a scientific experiment. This is very static information. By activating it, we can add some meta-information to it. For instance, the answer may be an AXML document, with some service calls to obtain details on the provenance of data and other calls to access existing annotations that were made on this experiment or to attach some new annotation. The metadata could also allow obtaining different versions of the same document and to access related information. The services invoked in the intensional part of the answer may be in charge of maintaining (possibly implicitly) the metadata and, in particular, the provenance information.

Context awareness and personalization A piece of information, be it extensional or intensional, may depend on some context. For instance, if a user obtained a list of restaurants, the content of the list may depend on predefined preferences, like the user's taste for certain kinds or food, or on religious restrictions. Also, the explicit presence of some details in this list may be guided by the device used by the user, possibly to accommodate limitations of her screen, or by the way the information will be used. For instance, all data should be extensional if the information is to be used off-line.

Enriching data This is the process of adding value to an answer by applying some external tools. The enrichment may occur at the level of the entire service answer or at a different granularity, e.g., an XML element. Typical enrichment may include links with the user context, extracting related concepts from an ontology, transforming from one type (XML schema) to another, or translating the answer to a preferred language. All these tasks can be uniformly abstracted as computations performed by dedicated Web services. These services can easily be leveraged in AXML answers, with the required granularity.

Physical motivations The motivations we presented above can be qualified as logical. It is interesting to note that most of them have counterparts at the physical level. We next illustrate some of these physical motivations.

To see one aspect that typically plays an important role on the performance and quality of Web servers, consider *Dynamicity*. Suppose some Web server S publishes an electronic literature catalog and that a comparative shopping site S' downloads the pages of the catalog regularly. The cache of S' will contain these pages, but information such as book prices that change rapidly will often be stale. The Web server S may decide to make catalog pages intensional, and represent the price information as service calls. Then, S' is able to refresh this price information, by invoking the corresponding service call, without having to reload the complete catalog page from S. This results in savings in communication. Furthermore, we will see in Chapter 6, that S' will even be able to *subscribe* to price changes and to be notified of such changes without the need of polling S. As a consequence, the cache of S' will have a more accurate copy of the catalog.

As another example, consider *Knowledge*. Knowledge may be exchanged in the form of intensional information, for instance to facilitate tasks such as routing. Suppose some system A requests some data from system B and B in fact finds the data in some system C. Then the data from C to A would transit via B. However, if B answers A intensionally, then A may obtain the data directly from C, which possibly results in saving in communication.

3.2.2.2 Calling with AXML parameters

We now turn to the benefits that can be obtained from using Web services that accept AXML documents as input parameters. In the presence of such services, the caller does not have to provide the input parameters extensionally, but can use the help of other services to construct these parameters. Therefore, the overall computation is not performed just by the invoked service, but may involve several participants.

Data-driven composition of services AXML parameters allow a simple kind of service composition, basically by connecting the outputs of some services to the input of others. For instance, if I want to know the phone number of Google's CEO, I can express that as a service call to a White Pages service, with an intensional parameter that is a call to Google's Company description service, that gives the name of the current CEO of the company. Also, service calls may appear with more than one level of nesting, i.e as parameters of service calls that appear in parameters of service calls, and so on recursively. As a simple example, we may want to know the phone number of the president of the country that currently has the presidency of the European Union (this changes every 6 month), using a service that gives the name of this country, composed with a service that gives the name of the president for a given country, and with the white pages service we used before.

More generally, intensional input parameters of services do not have to consist of a single call, but can be arbitrary AXML documents, therefore they can contain calls to several services, mixed with extensional XML data, and with an arbitrary depth of nesting for the service calls.

3.3 Related work

Active XML touches several areas in data management and Web computing. We next consider these topics and explain how the AXML model relates to them. Works that are related to the particular aspects of AXML that will be presented in the remainder of this thesis will be considered in the corresponding chapters.

Data integration Data integration systems typically consist of data sources, that provide information, and of mediators or warehouses, that integrate it with respect to a schema. The relationship between the former and the latter is defined *at the schema level*, using mapping rules between the schemas of sources and the integration schema that follow either a global-as-view or local-as-view approach [43, 64]. These mappings are used to translate queries on the integration schema into queries on the sources. Recent research works [67, 84] and commercial products [44, 83] focus on data integration for XML data, and the associated query and schema languages.

Significant attention was also recently devoted to "semantic mediation" [14, 51], where the integration involves an ontology. The main difference in this context is a richer data model for the integration schema, expressed in languages such as OWL [81], which are closer to object-oriented data models, with features such as inheritance. Formalisms for querying such ontologies are often variations of Description Logics [16] and different rewriting techniques are needed for data integration in that context [20].

By contrast to these schema-based and ontology-based approaches, Active XML favors links to external sources of information *directly in the data*. This approach is more dynamic, since new sources can be discovered in data, at run time, and is complementary to traditional schema-based query rewriting techniques, that tackle semantic heterogeneity between sources.

Moreover, in traditional data integration systems, sources typically allow *queries* to be asked to them. In AXML, sources can be more restrictive, in the sense that they only expose a particular set of function, with have a well-defined signature, exposed in their WSDL definition. These functions can be viewed as predefined parameterized queries on the source's data, with binding patterns that specify which variables (the input parameters) must be bound in order to enable their evaluation. But services can also be implemented using other means (e.g. as XSLT transformations, compositions of other services or programs) without any consequence on their use in the AXML model. A recent work [85] adopted an alternative approach, closer to traditional data integration, where Web services take queries as single input parameter, and their signature is used to restrict the set of allowed queries.

While mediators/warehouses can also serve as data sources for higher integration levels, a clear hierarchy between the data providers typically exists, which is defined at design time, by specifying the appropriate mapping rules. In contrast, the AXML model favors a more dynamic, ad hoc network topology, that corresponds mainly to the service calls embedded in AXML documents, which can evolve at run time, due to the presence of service calls in the answers of AXML services. Moreover, as we will see in Chapter 6, AXML systems (called peers) play a symmetric role, both as data sources and as partially materialized views over other peers, thus enabling a more flexible and scalable network architecture for data integration. Web services are used as uniform wrappers for heterogenous data sources, but also provide generic means to apply various transformations on the retrieved data. In particular, tools developed for schema/data translation and semantic integration, e.g. [64] can be wrapped as Web services and used to enrich the Active XML framework.

Intensional information The presence of service calls that can be materialized makes AXML documents *intensional*, since some of the data is given explicitly, whereas for some of it, a definition (i.e. the means to acquire it when needed) is given. Intensional information was mostly studied in a relational setting, in the context of Datalog, a language for deductive databases [9, 74]. There, the extensional database consists of regular relations, while the intensional database consists of a set of predicates defined as conjunctive queries over the (extensional and intensional) database. An important question in the context of intensional information is the one of lazy evaluation: given a query to evaluate, can one materialize (in our case, this corresponds to the invocation of service calls) the minimal amount of intensional information needed to answer the query? Algorithms were proposed for that in the relational setting, namely Magic Sets [17] and Query-SubQuery [99]. We will address the problem of lazy evaluation in the context of Active XML data in Chapter 5.

Services composition and integration The integration and composition of Web services has been an active field of research [101] recently. Before Web services appeared, intensional data was used in Multilisp [52], under the form of "futures", i.e., handles to results of ongoing computations, to allow for parallelism. Ambients [30, 29], as *bounded* spaces where computation happens, also provide a powerful abstraction for processes and devices mobility on the Web.

More recently, standard languages have even been proposed in the industry (like IBM's Web Services Flow Language [105] or Microsoft's XLang [108], that converged into the BPEL4WS proposal[25]), for specifying how Web services interact with each other and how messages can be passed consistently between business processes through service interfaces. An overview of existing work with a good attempt to formalize and classify the problems can be found in [57].

While AXML also allows to compose services, by using the output of some service calls as input to other ones, and even to define new services based on such (possibly recursive) compositions, our focus is not on workflow and process-oriented techniques [32] but on *data*. In other words, AXML is not a framework for service composition, but for distributed data management using Web services.

Embedded calls As already mentioned, the idea of embedding function calls in data is not a new one. Embedded functions are already present in relational systems [98], e.g., as stored procedures. Stonebraker et al. even proposed to use them as a first-class datatype [93]. Of course, method calls form a key component of object databases [31].

In the Web context, scripting languages such as php [86] or JSP [62] have made popular the integration of processing inside HTML or XML documents. Combined with standard database interfaces such as JDBC and

ODBC, functions are used to integrate results of queries (e.g., SQL queries) into documents. A representative example for this is Oracle XSQL [80].

The introduction of service calls in XML documents is thus very natural. Indeed, external functions were present in Lore [56] and an extension of object databases to handle semistructured data is proposed in [82], thereby allowing to introduce external calls in XML data. Our work is tailored to XML and Web services. In that sense, it is more directly related to Microsoft Smart Tags [87], where service calls can be embedded in Office documents, mainly to enrich the user experience by providing contextual tools, while our goal is to provide means of controlling and enriching the use of Web service calls for data management purposes, in a distributed setting. Other systems recently proposed languages based on XML with embedded calls to Web services, like Macromedia MX [65], or Apache Jelly [60]. The techniques we propose for AXML in Chapters 4 and 5 are indeed applicable to these languages.

Active databases and triggers The activation of service calls is closely related to the use of triggers [98] in relational databases, or rules in active databases [102]. Active rules were recently adapted to the XML/XQuery context [23]. A recent work considered firing Web service calls [24]. We harness these concepts to obtain a powerful data integration framework based on Web services. In some sense, the present work is a continuation of previous works on *ActiveViews* [3]. There, declarative specifications allowed for the automatic generation of applications where users could cooperate via data sharing and change control. The main differences with ActiveViews are that (i) AXML promotes peer-to-peer relationships vs. interactions via a central repository, and (ii) the cornerstones of the AXML language are XPath, XQuery and Web services vs. object databases [31].

Peer-to-Peer computing is gaining momentum as a large-scale resource sharing paradigm by promoting direct exchange between equal-footed peers [63, 75]. We propose a system where interactions between peers are at the core of the data model, through the use of service calls. Moreover, it allows peers to play different roles, and does not impose strong constraints on interaction patterns between peers, since they are allowed to define and use arbitrary Web services. While we do not consider issues such as dynamic data placement and replication, or automatic peer discovery, we believe that solutions developed in the peer computing community for these problems (see, for instance [75]) can benefit our system, and plan to investigate that in future work.

3.4 Thesis organization

We saw the main components of the Active XML model, namely AXML documents, that are XML documents with embedded calls to Web services, and AXML services, i.e. Web services that exchange AXML documents, and some important motivations for the AXML model.

We will next examine some of the novel important issues raised by the AXML model:

- First, we will consider, in Chapter 4 the issue of controlling the exchange of AXML documents among systems. We will adopt a type-based approach to control which calls needs to be materialized before a document is exchanged, and rely on automata-based techniques to enforce these types.
- Then, we will propose in Chapter 5 a technique that is capital for querying AXML documents, that automatically determines which service calls should be materialized in order to answer a given query. This lazy evaluation method will be based on query rewriting and analysis techniques, and we will also propose several important optimizations.

In the last part of the thesis, we will present the infrastructure we built to support the AXML model and ease the development of AXML applications. This infrastructure adopts a peer-to-peer paradigm, in the sense that each participating system acts both as a client and as a server. We will study the functionalities of AXML peers and illustrate the convenience of this paradigm through a number of applications we built, and conclude.

Chapter 4

Exchanging AXML data

In the previous chapter, we illustrated some of the benefits and issues involved in exchanging AXML documents in a distributed setting. Indeed, getting or sending some information extensionally or intensionally does matter. The goal of this chapter is to present some efficient techniques that can be provided to systems in order to control whether they exchange information intensionally or extensionally. More precisely, given a document to exchange, our goal is to control which parts of the document should be materialized, and which should not.

Before getting to the description of the technical solution we propose, let us first see some of the considerations that may guide the choice of materializing some intensional data or not. Of course, all the motivations we gave in the previous chapter for AXML documents and services are important to consider. We focus here on considerations that are particular to the context of exchanging information.

Performance The decision whether to execute calls before or after the data transfer may be influenced by the current system load or the cost of communication. For instance, if the sender's system is overloaded or communication is expensive, the sender may prefer to send smaller files and delegate as much materialization of the data as possible to the receiver. Otherwise, it may decide to materialize as much data as possible before transmission, in order to reduce the processing on the receiver's side.

Capabilities Although Web services may in principle be called remotely from everywhere on the Internet, it may be the case that the particular receiver of the intensional document cannot perform them, e.g., a newspaper reader's browser may not be able to handle the intensional parts of a document. And even if it does, the user may not have access to a particular service, e.g., by lack of access rights. In such cases, it is compulsory to materialize the corresponding information before sending the document.

Security Even if the receiver is capable of invoking service calls, she may prefer not to do so for security reasons. Indeed, service calls may have side effects. Receiving intensional data from an untrusted party and invoking the calls embedded in it may thus lead to severe security violations. To overcome this problem, the receiver may decide to refuse documents with calls to services that do not belong to some specific list. It is then the responsibility of a helpful sender to materialize all the data generated by such service calls before sending the document.

Functionalities Last but not least, the choice may be guided by the application. In some cases, e.g. for a UDDI-like service registry, the origin of the information is what is truly requested by the receiver, hence service calls should not be materialized. In other cases, one may prefer to hide the true origin of the information, e.g., for confidentiality reasons, or because it is an asset of the sender, so the data must be materialized. Finally, calling services might also involve some fees that should be payed by one or the other party.

Observe that the data returned by a service may itself contain some intensional parts. As a simple example, *TimeOut* may return a list of ten exhibits, along with a service call to get more. Therefore, the decision of materializing some information or not is inherently a recursive process. For instance, for clients who cannot handle intensional documents, the newspaper server needs to recursively materialize all the document before sending it.

How can one guide the materialization of data? For purely extensional data, schemas (like DTD and XML Schema) are used to specify the desired format of the exchanged data. Similarly, we use schemas to control the exchange of intensional data and, in particular, the invocation of service calls. The novelty here is that schemas also entail information about which parts of the data are allowed to be intensional and which service calls may appear in the documents, and where. Before sending information, the sender must check if the data, in its current incarnation, matches the schema expected by the receiver. If not, the sender must perform the required calls for transforming the data first, if this is possible.

A typical such scenario is depicted in Figure 4.1. The sender and the receiver, based on their personal policies, have agreed on a specific data exchange schema. Now, consider some particular data t to be sent (represented by the grey triangle in the figure). In fact, this document represents a set of equivalent, increasingly materialized, pieces of information – the documents that may be obtained from t by materializing some of the service calls (q, g and f). Among these documents, the sender must find at least one that conforms to the exchange schema (e.g., the dashed one) and send it.

This schema-based approach is particularly relevant in the context of Web services, since their input parameters and their results must match particular XML Schemas, that are specified in their WSDL descriptions. The techniques presented in this chapter can be used to achieve this.

The contributions of this chapter are as follows:

- We provide a simple but flexible XML-based syntax to embed service calls in XML documents. We introduce an extension of XML Schemas for describing the required structure of the exchanged data. This consists in adding new type constructors for service call nodes. In particular, our typing distinguishes between accepting a concrete type, e.g. a *temperature* element, and accepting a service call returning some data of this type, e.g., $() \rightarrow temperature$.

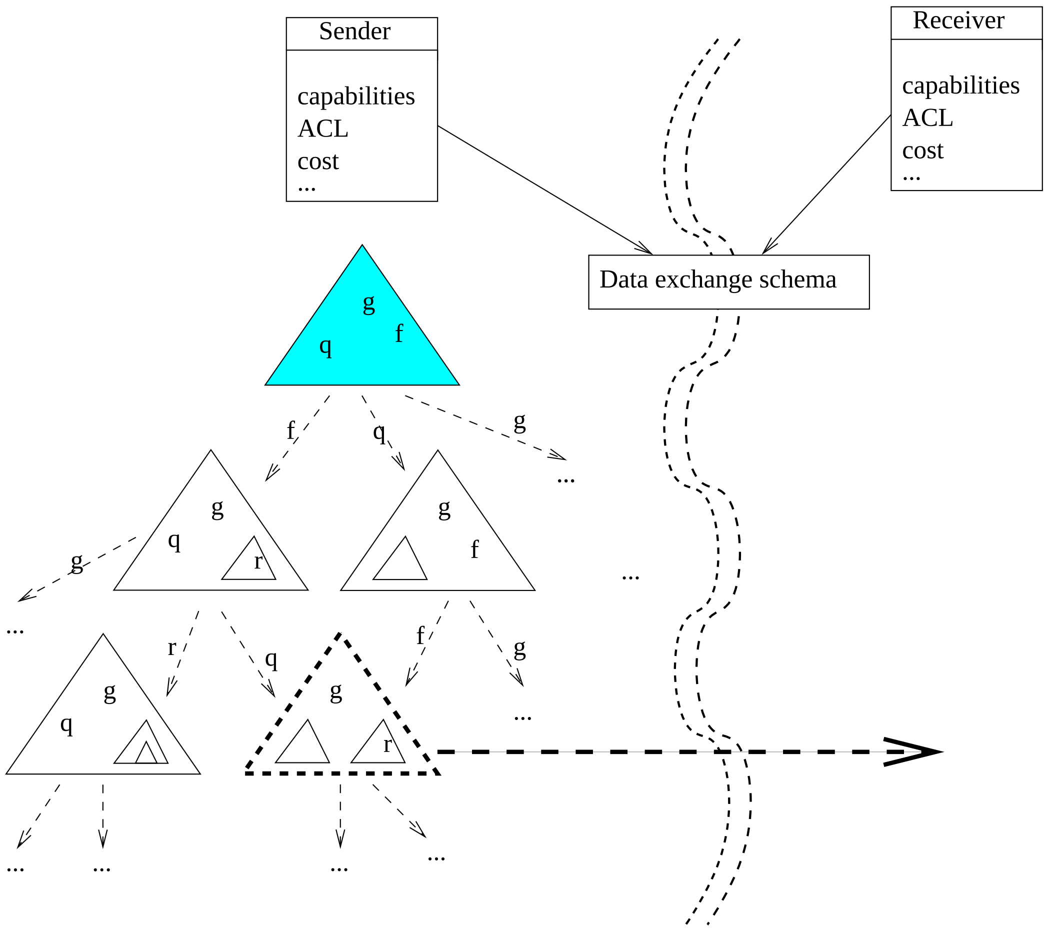

Figure 4.1: Data exchange scenario for intensional documents.

- Given a document t and a data exchange schema, the sender needs to decide which data has to be materialized. We present algorithms that, based on schema and data analysis, find an effective sequence of call invocations, if such a sequence exists (or detect a failure if it does not). The algorithms provide different levels of guarantee of success for this rewriting process, ranging from "sure" success to a "possible" one.

- At a higher level, in order to check compatibility between applications, the sender may wish to verify that *all* the documents generated by its application may be sent to the target receiver, which involves comparing two schemas. We show that this problem can be easily reduced to the previous one.

As explained above, our algorithms find an effective sequence of call invocations, if one exists, and detect failure otherwise. In a more general context, an error may arise because of type discrepancies between the caller and the receiver. One may then want to modify the data and convert it to the right structure, using data translation techniques such as [34, 38]. As a simple example, one may need to convert a temperature from Celsius degrees to Fahrenheit. In our context, this would amount to plugging (possibly automatically) intermediary external services to perform the needed data conversions. Existing data conversion algorithms can be adapted to determine when conversion is needed. Our typing algorithms can be used to check that the conversions lead to matching types. Data conversion techniques are complementary and could be added to our framework. But the focus here is on partially materializing the *given data* to match the specified schema.

The core technique of this work is based on automata theory. To simplify the presentation, we first detail a simplified version of the main algorithm. We then describe a more dynamic, optimized one, that is based on the same core idea and is used in our implementation.

Although the problems studied here are related to standard typing problems in programming languages [73], they differ here due to the regular expressions present in XML schemas. The general problem that was formalized in [70] was recently shown to be undecidable in [76]. We introduce a restriction that is practically founded, and leads to a tractable solution.

All the ideas presented here have been implemented and tested in the context of the system presented in Chapter 6. The algorithms described here were used to implement a module that controls the types of documents being sent to (and returned by) the Web services provided by the system. This module is in charge of materializing the appropriate data fragments to meet the interface requirements.

This chapter is organized as follows: Section 4.1 describes a simple data model and schema specification language and formalizes the general problem. Additional features for a richer data model that facilitate the design of real life applications are also introduced informally. Section 4.2 focuses on difficulties that arise in this context, and presents the key restriction that we consider. It also introduces the notions of "safe" and "possible" rewritings, which are studied in Section 4.3 and 4.4 respectively. The problem of checking compatibility between intensional schemas is considered in Section 4.5. The implementation is described in Section 4.6. A last section mentions works related to our typing algorithms and concludes.

4.1 The model and the problem

To simplify the presentation, we start by formalizing the problem using a DTD-like schema specification. More precisely, we define the notion of *rewriting*, which corresponds to the process of invoking some service calls in an AXML document, in order to make it conform to a given schema. Once this is clear, we explain how things can be extended to provide the features ignored by the first simple model, and in particular we show how richer schemas are taken into account.

4.1.1 The simple model

We first define *schemas*, before formalizing the key notion of *rewritings*, and stating the results obtained in this setting, which will be detailed in the following sections.

Simple schemas

We next define simple DTD-like schemas for intensional documents. The specification associates (1) a regular expression with each element name, that describes the structure of the corresponding elements, and (2) a pair of regular expressions with each function name, that describe the function signature, namely its input and output types

Definition 4.1.1 *A document schema is an expression (L, F, τ), where $L \subset \mathcal{L}$ and $F \subset \mathcal{F}$ are finite sets of labels and function names respectively, τ is a* *function that maps each label name $l \in L$ to a regular expression over $L \cup F$ or to the keyword "data" (for atomic data), and maps each function name $f \in F$ to a pair of such expressions, called the input and output type of f and denoted by $\tau_{in}(f)$ and $\tau_{out}(f)$.*

We will talk about the schema τ whenever L and F are understood. For instance, the following is an example of a schema τ:

data :

$$
\begin{array}{lcl}
\tau(newspaper) & = & title.date.(Get\_Temp \mid temp).(TimeOut \mid exhibit^*) \\
\tau(title) & = & data \\
\tau(date) & = & data \\
\tau(temp) & = & data \\
\tau(city) & = & data \\
\tau(exhibit) & = & title.(Get\_Date \mid date)
\end{array}
$$

functions :

$$
\begin{array}{lcl}
\tau_{in}(Get\_Temp) & = & city \\
\tau_{out}(Get\_Temp) & = & temp \\
 & & \\
\tau_{in}(TimeOut) & = & data \\
\tau_{out}(TimeOut) & = & (exhibit \mid performance)^* \\
 & & \\
\tau_{in}(Get\_Date) & = & title \\
\tau_{out}(Get\_Date) & = & date
\end{array}
$$

We next define the semantics of a schema, i.e., the set of its instances. Recall from Section 3.1.2 that $\mathcal{L}, \mathcal{F}, \mathcal{D}$ are the domains of labels, function names, and data values respectively. If R is a regular expression over $\mathcal{L} \cup \mathcal{F}$, we denote by $lang(R)$ the regular language defined by R. The expression $lang(data)$ denotes the set of data values in $\mathcal{D}$.

Definition 4.1.2 *An AXML document t is an instance of a schema $s = (L, F, \tau)$ if for each data node (resp. function node) $n \in t$ with label $l \in L$*

(resp. $l \in F$), the labels of n's children form a word in $lang(\tau(l))$ (resp. in $lang(\tau_{in}(l))$).

For a function name $f \in F$, a sequence $t_1, \ldots, t_n$ of AXML trees is an input instance (resp. output instance) of f, if the labels of the roots form a word in $lang(\tau_{in}(f))$ (resp. $lang(\tau_{out}(f))$, and all the trees are instances [1] *of s.*

It is easy to see that the document of Figure 3.3.a is an instance of the schema τ, but not of a schema τ' identical to τ above, except for:

$$\tau'(newspaper) = title.date.temp.(TimeOut \mid exhibit^*)$$

However, since $\tau_{out}(GetTemp) = temp$, the document can always be turned into an instance of the schema τ', by invoking the $GetTemp$ service call and replacing it by its return value. On the other hand, consider a schema τ'' identical to τ, except for:

$$\tau''(newspaper) = title.date.temp.exhibit^*$$

According to its signature, a call to $TimeOut$ may also return *performance* elements. Therefore, in general, the document may not become an instance of the schema τ''. However, it is *possible* that it becomes one (if $TimeOut$ returns a sequence of *exhibits*). The only way to know is to call the service.

This type of "on-line" testing is acceptable if the calls have no side effects or do not cost money. If they do, we might want to warn the sender, before invoking the call, that the overall process may not succeed, and see if she wants to proceed anyway.

Rewritings

When the proper invocation of service calls leads for sure to the desired structure, we say that the rewriting is *safe*, and when it only possibly does, that this is a *possible* rewriting. These notions are formalized next.

[1] Like in DTDs, every subtree conforms to the same schema as the whole document.

Definition 4.1.3 *For an AXML document t, we say that $t \xrightarrow{v} t'$ if t' is obtained from t by selecting a function node v in t with some label f and replacing it by an arbitrary output instance of f [2]. If $t \xrightarrow{v_1} t_1 \xrightarrow{v_2} t_2 \ldots \xrightarrow{v_n} t_n$ we say that t rewrites into t_n, denoted $t \xrightarrow{*} t_n$. The nodes $v_1, \ldots, v_n$ are called the rewriting sequence. The set of all trees t' s.t. $t \xrightarrow{*} t'$ is denoted $ext(t)$.*

Note that in the rewriting process, the replacement of a function node v by its output instance is independent of any function semantics. In particular, we may replace two occurrences of the same function call (i.e. invoking the same function, with the same parameters) by two different output instances. Stressing somewhat the semantics, this can be interpreted as if the value returned by the function changes over time. This captures the behavior of real life Web services, like a temperature or stock exchange service, where two consecutive calls may return a different result.

Definition 4.1.4 *Let t be a tree and s be a schema. We say that t possibly rewrites into s if $ext(t)$ contains some instance of s. We say that t safely rewrites into s either if t is already an instance of s, or if there exists some node v in t s.t. all trees t' where $t \xrightarrow{v} t'$ safely rewrite into s.*

The fact that t safely rewrites into s means that we can be sure, without actually making *any* call, that we can choose a sequence of calls that will turn t into an instance of s. For instance, the document of Figure 3.3.a *safely rewrites* into the schema τ' but only *possibly rewrites* into τ''.

Finally, to check compatibility between applications, we may want to check whether *all* documents generated by one application (e.g. the sender application) can be safely rewritten into the type accepted by the second application.

Definition 4.1.5 *Let s be a schema with some distinguished label r called the root label. We say that s safely rewrites into another schema s' if all the instances t of s with root label r rewrite safely into instances of s'.*

[2] By *replacing the node by an output instance* we mean that the node v and the subtree rooted at it are deleted from t, and the forest trees $t_1, \ldots, t_n$ of some output instance of f are plugged at the place of v (as children of v's parent).

For instance, consider the schema τ presented above with *newspaper* as the root label. This schema safely rewrites into the schema τ' but does not safely rewrite into τ''.

The results

Going back to the data exchange scenario described in the introduction, we can now specify our main contributions:

- We present an algorithm that tests whether a document t can be safely rewritten into some schema s and, if so, provides an effective rewriting sequence.

- When safe rewriting is not possible, we present an algorithm that tests whether t may be possibly rewritten into s, and finds a successful rewriting sequence, if one exists.

- We also provide an algorithm for testing, given two schemas, whether one can be safely rewritten into the other.

4.1.2 A Richer Data Model

In order to make our presentation clear, and to simplify the definition of document and schema rewritings, we used so far a very simple data model and schema language. We will now present some useful extensions that bring more expressive power, and facilitate the design of real life applications.

Function patterns The schemas we have seen so far specify that a *particular function*, identified by its name, may appear in the document. But sometimes, one does not know in advance which functions will be used at a given place, and yet may want to allow their usage, provided that they conform to certain conditions. For instance, we may have several editions of the *newspaper* of Figure 3.3.a, for different cities. A common intensional schema for such documents should not require the use of a particular *Gettemp* function, but rather allow for a *set* of functions, which have a suitable signature: They should accept as single parameter a *city* element, and return a *temperature* element, as previously defined in τ. The particular weather forecast service that will be used may depend on the city and be, for instance, retrieved from some UDDI service registry. One may also want to enforce some security policies, e.g. be allowed to specify that the allowed functions should return only extensional results.

To specify such sets of functions, we use *function patterns*. A function pattern definition consists of a boolean predicate over function names and a function signature. A function matches the pattern if its name satisfies the boolean predicate and its signature is the same as the required one. A more liberal definition would be to require that the function signature only be *subsumed* by the one specified in the definition. This is possible but is computationally more intense, since it entails checking inclusion of tree languages. We will not do it here.

In terms of implementation, one can assume that this new boolean predicate is implemented as a Web service that takes a function name as input and returns true or false.

To take this feature into account in our model, we define $\mathcal{P}$ to be a domain of function pattern names. A schema $s = (L, F, P, \tau)$ now also contains, in addition to the elements and functions, a set of function patterns $P \subset \mathcal{P}$. τ associate with each function pattern $p \in P$ a signature and a boolean predicate over function names. We can now, for instance, write a schema for our local newspapers as:

$$\begin{aligned}
&\tau(newspaper) = title.date.(Forecast \mid temp).(TimeOut \mid exhibit^*)\\
&\tau_{name}(Forecast) = UDDIF \wedge InACL\\
&\tau_{in}(Forecast) = city\\
&\tau_{out}(Forecast) = temp
\end{aligned}$$

This schema enforces the fact that the function used in the document has the proper signature and satisfies the boolean predicates $UDDIF$ and $InACL$. The first predicate ($UDDIF$) is a Web service that checks if the given function (service) is registered in some particular UDDI registry. Predicate $InACL$ then verifies if the caller has the necessary access privileges for executing the given function (calling the service). More generally, any Web service that allows the verification of some property of the particular function node in the document (here, the weather forecast service), possibly with respect to some contextual information (e.g. the identity of the caller, the system date, etc.) can be used.

Wildcards Together with function patterns, one may also use wildcards in schemas. Their use is already common for *data*. In XML Schema, the keyword *any* expresses the fact that a certain part of a document may contain an arbitrary element, attribute, or even an unconstrained subtree. XML Schema further allows one to restrict wildcards to (or exclude from them) certain domains of data, based on their *namespace* [77]. This extends naturally to our context. We consider the namespace of a function node in an intensional document to be the namespace of the called Web service [3]. Therefore, we can use wildcards to allow certain document parts to contain arbitrary sub-trees with arbitrary functions, or restrict them to (resp. exclude from them) certain classes of functions.

We believe that the combination of wildcards and function patterns provides a good level of flexibility to describe the structure of documents. For instance, one may specify that the temperature is obtained from an arbitrary function that returns a correct *temp* element, but may take any argument, being data or function call.

Restricted service invocations We assumed so far that all the functions appearing in a document may be invoked in a rewriting, in order to match a given schema. This is not always the case, for the same reasons as mentioned in the introduction (security, cost, access rights, etc.). The logic of rewritings will have to take this into account, essentially by considering, among all possible rewritings, only a proper subset. For that, the function names/patterns in the schema can be partitioned into two disjoint groups of *invocable* and *non-invocable* ones. A *legal* rewriting is then one that invokes only invoca-

[3] Which is described in its WSDL description and, in our model, is one of the components of the function name.

ble functions. The notions of safe and possible rewritings extend naturally to consider only legal rewritings. Since we are interested here only in such rewritings, whenever we talk in the sequel about a function invocation, we mean an invocable one.

XML, XML Schema and WSDL We ignored, in both the data model of Section 3.1.2 and the schema language defined above, a number of features of XML, such as attributes, and used a single domain for data values. A richer setting may be obtained by using the full fledged XML data model [109]. Similarly, richer schemas may be defined by adopting XML Schema [110], rather than using the simple DTD-like language used above. Indeed, our implementation is based on the full XML model and on an extension of XML Schema.

In our prototype, function calls embedded in XML documents are represented by special function elements that identify the Web services to be invoked and specify the value of input parameters. XML Schemas are enriched for intensional documents (to form XML Schema$_{int}$) by function and function pattern definitions. The "int" subscript stands for intensional, since our types can explicitly specify that some data is intensional. In both cases, things are very much along the lines of the simple model we used above. We will see an example and more details of this in Section 4.6.

Function signatures are usually specified by service providers as WSDL definitions. We similarly extend WSDL to allow the use of XML Schema$_{int}$ instead of just XML Schema for type specifications, and we term this extended language WSDL$_{int}$.

While intensional XML documents use a standard XML syntax, XML Schema$_{int}$ schemas do not comply with the XML Schema syntax. The extension is minimal, and very much along the lines of the simple syntax we used above. We will also see an example and more details in Section 4.6. Note that this is not the case for WSDL, since its specification does not enforce the use of a specific schema language. Therefore WSDL$_{int}$ documents are valid WSDL documents.

4.2 Exchanging Intensional Data

We start by considering document rewriting. Schema rewriting is considered later in Section 4.5.

Given a document t that the sender wishes to send, and a data exchange schema s, the sender needs to rewrite t into s. A possible process is the following:

1. Check if t safely rewrites to s and if so, find a *rewriting sequence*, namely a sequence of functions that need to be invoked to transform t into the required structure.

2. If a safe rewriting does not exist, check whether at least t *may* rewrite to s. If it is acceptable to do so (the sender accepts that the rewriting may fail), try to find a successful rewriting sequence if one exists.

A variant is to combine safe and possible rewritings. For instance, one could consider a mixed approach, that first invokes some function calls, and then attempts from there to find safe rewritings. There are many alternative strategies.

Also, one should note that, for both safe and possible rewritings, there may be more than one adequate rewriting sequence. The system should choose the shortest or cheapest one, according to some criteria. The optimisation problem of selecting the "best" (safe or possible) rewriting is orthogonal to the one we are considering, and will not be studied here.

We will first consider safe document rewriting, then move to possible rewriting, and finally consider the mixed approach. As in the previous section, to simplify the presentation, we first consider the problems in the context of the simple data model defined above. Then in Section 4.6 we will show that the proposed solutions naturally extend to richer data/schemas and in particular to the context of full fledged XML and XML Schema.

Before presenting solutions, let us first explain some of the difficulties that one encounters when attempting to rewrite a document to a desired exchange schema. While the examples given in the previous sections were rather simple

– and one could determine by a simple observation of the document which service calls need to be materialized – things may be much more complex in general. We explain next why this is the case and present a restriction that will make the problem tractable.

Going back and forth The rewriting sequence may depend on the answers being returned by the functions: we may call one function at some place in the document, and then decide, possibly based on its answer, that another function in the new data or in a different part of the document needs to be called, and so on. In general, this may force us to analyze the same portion of the document many times, re-examining the same function call again and again, deciding at each iteration whether, based on the answers returned so far, the function now needs to be called or not. Such an iterative process may naturally be very expensive. We thus restrict our attention here to a simpler class of "one-pass" *left-to-right* rewritings [4], where for each node, the children are processed from left to right, and once a child function is invoked, no further invocations are applied to its left hand sibling functions (i.e. successive children invocations are limited to the new children functions possibly returned by the call, plus the right hand siblings.). This restriction also applies to the results of function calls, which are also processed in a left-to-right manner.

Observe that in general, with this restriction, one can miss a successful rewriting that is not left-to-right. In all real-life examples we considered, left-to-right rewritings turned out not to be limiting.

Infinite search space The essence of safe rewriting is that it succeeds no matter what specific answers, among the possible ones, the invoked functions return. The domain of the possible answers of each function is determined by its output type. Since the regular expression defining this type may contain starred ("*") sub-expressions, this domain is infinite, and the safe rewriting should account for each possible element in this infinite domain. Moreover, the result of a service call may contain intensional data, namely

[4] One could choose similarly right-to-left.

other function calls. In general the number of such new functions may be unbounded. For instance, consider a *GetExhibits* function, with output type

$$\tau_{out}(GetExhibits) = GetExhibit^*.$$

When *GetExhibits* is invoked, an arbitrarily large number of *GetExhibit* functions may be returned. In order to investigate the entire search space, one would have to check for each of the occurrences whether this particular function call needs to be invoked and whether, after the invocation, the document can still be (safely) rewritten into the desired schema.

Recursive calls As explained above, when a function is invoked, the returned data may itself contain new calls. To conform to the target schema, these calls may need to be triggered as well. The answer again may contain some new calls, etc. This may lead to infinite computations. Observe that such recursive situations do occur in practice. For example, a search engine Web service may return, for a given keyword, some document URLs plus (possibly) a function node for obtaining more answers. Calling this function, one can obtain a new list and perhaps another function node, etc. If the target schema requires plain XML data we need to repeatedly call the functions until all the data has been obtained. In this example, and often in general, one may want to bound the recursion. This suggests the following definition and our corresponding restriction:

Definition 4.2.1 *For a rewriting sequence $t \xrightarrow{v_1} t_1 \ldots \xrightarrow{v_n} t_n$, we say that a function node v_j depends on a function node v_i if $v_j \in t_i$ but $\notin t_{i-1}$ (namely if the node v_j was returned by the invocation of the function v_i).*

We say that a rewriting sequence is of depth k if the dependency graph among the function nodes considered in the sequence contains no paths of length greater than k.

The restriction We restrict our attention to k-depth left-to-right rewritings.

Note that while this restriction limits the search space, it still remains infinite, due to the starred sub-expressions appearing in the schema. However, under this restriction, we can exhibit a finite representation (based on automata) of the search space and use automata-based techniques to solve the safe rewriting problem.

Even with this restriction, the framework is general enough to handle practical cases. The problem of arbitrary safe rewriting (without the left-to-right k-depth restriction) was recently shown to be undecidable [76]. Further work by the same authors [76, 91] shows that the left-to-right safe rewriting problem is actually decidable, without the k-depth restriction, but the corresponding algorithms have a much higher complexity (EXPTIME or 2EXPTIME, depending on whether the target language is deterministic or not) — thus are mostly of theoretical interest.

4.3 Safe Rewriting

In this section, we present an algorithm for k-depth safe rewriting.

We are given a document tree t and a schema $s_0 = (L_0, F_0, \tau_0)$ describing the signatures of all the functions in the document (as well as the elements/functions used in these signatures). This corresponds to having a WSDL description for each service being used, which is a normal requirement for Web services. We are also given a data exchange schema $s = (L, F, \tau)$, and our goal is to safely rewrite t into s (with a k-depth left-to-right rewriting).

To simplify, we assume that function types are the same in s_0 and s, including definitions of the corresponding sub-elements. This is reasonable since the function definitions represent the WSDL description of the functions, as given by the service providers. While this assumption simplifies the rewriting process, it is not essential. The algorithm can be extended to handle distinct signatures. Note that this is not a constraint on the document t, since it does not have to conform to s_0.

For clarity, we decompose the presentation of the algorithm into three parts.

1. The first part explains how to deal with function parameters. The main point is that, since the parameters may themselves contain other function calls (with parameters), the tree rewriting starts from the deepest function calls and recursively moves upward.

2. The second part explains how the rewriting in each such iteration is performed. The key observation is that this can be achieved by traversing the tree from top to bottom, handling one node (and its direct children) at a time.

3. Finally, the third and most intricate part, explains how each such node and its direct children are handled. In particular, we show how to decide which of the functions among these children need to be invoked in order to make the node fit the desired structure.

For presentation reasons, we give here a simplified version of the actual algorithm used in the implementation. To optimize the computation, a more dynamic variant, based on the same idea, is used there. We explain the main principles of this variant in Section 4.6.

4.3.1 Rewriting function parameters

Should a function node be materialized or not, its parameters should be of the right type. In the first case, this is necessary for the function call to be valid. In the second case, this is a requirement of the schema, in order for the function node to be of the proper type. If the parameters are not of the right type, they should be rewritten to fit that type. When rewriting the parameters, again, the functions appearing in them can be invoked only if their own parameters are (or can be rewritten into) the expected input type. We thus start from the "deepest" functions, i.e. those having no function occurrences in the parameters, and recursively move upward:

- For the deepest functions, we verify that their parameters are indeed instances of the corresponding input types. If not, the rewriting fails.

- Then moving upward, we look at a function f and its parameters. All the functions appearing in these parameters were already handled - namely their parameters can be safely rewritten to the appropriate type. We thus ignore the parameters of these lower level calls (together with all the functions included in them) and just try to safely rewrite f's own parameters into the required structure. If this is not possible, the rewriting fails, for the same reason as above.

At the end of this process, we know that all the outermost function calls in t have the desired property. We can thus ignore their parameters and need to safely rewrite t into s by invoking only its outermost calls.

4.3.2 Top down traversal

In each iteration of the above recursive procedure we are given a tree (or a forest) where the parameters of all the outermost functions have already been handled, and we need to safely rewrite the tree (resp. forest) by invoking only these outermost functions. To do that, we can traverse the tree (resp. forest) top down, treating at each step a single node and its immediate children.

Consider a node n whose children labels form a word w. Note that the subtree rooted at n can be safely rewritten into the target schema $s = (L, F, \tau)$ if and only if (1) w can be safely rewritten into a word in $lang(\tau(label(n))$, and (2) each of n's children subtrees can itself be safely rewritten into an instance of s . Note that since we assumed that s_0 and s agree on function types, we only need to rewrite the *original* children of n and not those that are returned by function invocations. Therefore, we can start from the root and, going down, for each node n try to safely rewrite the sequence of its children into a word in $lang(\tau(label(n)))$. The algorithm succeeds if all these individual rewritings succeed.

The fact that we are processing the tree top-down allows us to handle sub-typing. Indeed, XML Schema allows elements of the same label to have different types, but guarantees that the type of an element is uniquely determined by its label, its preceding siblings, and the type of its parent. Therefore a top-down traversal is completely deterministic. Note that for the simple

DTD-like language we are using, a bottom-up validation would also be a correct choice. In this case, the previous phase of parameter rewriting can be merged with the tree traversal, since the function nodes are guaranteed to be encountered in the right order, i.e. the deepest ones first, and the outermost ones last.

The safe rewriting of a word w involves the invocation of functions in w and (recursively) new functions that are added to w by those invocations. To conclude the description of our rewriting algorithm we thus only need to explain how this is done.

4.3.3 Rewriting the children of a node n

This is the most intricate part of the algorithm. We are given a word w - the sequence of labels of n's children - and our goal is to rewrite w to fit the target schema. Namely, we need to rewrite w so that it becomes a word in the regular language $R = \tau(label(n))$. The rewriting process invokes functions in w and (recursively) new functions that are added to w by those invocations. Each such invocation changes w, replacing the function occurrence by its returned answer. The possible changes that the invocation of a function f_i may cause are determined by the output type $R_{f_i} = \tau_{out}(f_i)$ of f_i [5]. For instance, if $w = a_1, a_2, \ldots, f_i, \ldots, a_m$, invoking f_i changes w into some $w' = a_1, a_2, \ldots, b_1, \ldots, b_k, \ldots, a_m$ where $b_1, \ldots, b_k \in lang(R_{f_i})$.

Since the functions signatures, as well as the target schema, are given in terms of regular expressions, it is convenient to reason about them, and about the overall rewriting process, by analyzing the relationships between their corresponding finite state automata. We assume some basic knowledge of regular languages and finite state automata, and use in our algorithm standard notions such as the intersection and complement of regular languages and the Cartesian product of automata. For basic material, see for instance [53].

Given the word w, the output types $R_{f_1}, \ldots, R_{f_n}$ of the available functions, and the target regular language R, the algorithm in Figure 4.2 tests if

[5] Recall from the discussion above that the input parameters can be ignored.

safe rewriting (word w, functions output types $R_{f_1} \dots R_{f_n}$, target language R)

Build finite state automata for the following regular languages:
(a) An automaton A_w accepting w as a single word.
(b) Automata A_{f_i}, $i = 1 \dots n$, each accepting the regular language R_{f_i}. These automata should be deterministic.
(c) An automaton $\overline{A}$ accepting the complement of the regular language R. The automaton should be deterministic and *complete*, namely each state has outgoing edges for all possible letters.
Let $A_w^k := A_w$.
For $j = 1, \dots, k$
Consider all the edges $e = (v, u)$ in A_w^k that are labeled by an (invocable) function name f_i and were not treated in previous iterations. For each such edge:
(a) extend A_w^k by attaching a copy of the automaton A_{f_i}, with its initial and final accepting states linked to v and u resp. by ϵ moves.
(b) Denote v as a *fork* node (for the edge e).
(c) The two *fork options* of v (for e) are e itself and the new outgoing ϵ edge.
Construct the Cartesian product automaton $A_\times = A_w^k \times \overline{A}$.
The *fork nodes* and *fork options* in $A_\times$ reflect those of A_w^k:
(a) the *fork nodes* $[q, p] \in A_\times$ are those where q was a fork node in A_w^k.
(b) Similarly, a *fork option* in $A_\times$ consists of all edges originating from one fork option edge in A_w^k.
Mark nodes in $A_\times$ as follows.
(a) First mark all accepting states, (namely nodes $[q, p]$ where q and p are accepting states in A_w^k and $\overline{A}$ resp.
(b) Then iteratively: mark regular (non fork) nodes if one of their outgoing edges points to a marked node; mark fork nodes if in both their fork options (for some f_i) contain an edge that points to a marked node.
Return **true** if the initial state is marked, **false** otherwise.

Figure 4.2: Safe rewriting of w into R.

w can be safely rewritten into a word in R. Then, if the answer is positive, the algorithm presented in Section 4.3.4 finds a safe rewriting sequence.

We give the intuition behind this first algorithm next. To illustrate, we use the newspaper document in Figure 3.3.a. Assume that we look at the root *newspaper* node. Its children labels form the word $w = title.date.GetTemp.TimeOut$. Assume that we want to find a safe rewriting for this word into a word in the regular language $\tau'(newspaper)$ of the schema τ', namely

$$R = title.date.temp.(TimeOut \mid exhibit^*).$$

The process of rewriting involves choosing some functions in w and replacing them by a possible output; then choosing some other functions (which might have been returned by the previous calls) and replacing them by their output, and so on, up to depth k. For each function occurrence we have two choices: either to leave it untouched, or to replace it by some word in it output type. The automaton A^k_w constructed in Steps 5-10 of the algorithm represents precisely all the words that can be generated by such a k-depth rewriting process. The *fork* nodes are the nodes where a choice (i.e. invoking the function or not) exists, and the two *fork options* represent the possible consequent steps in the automaton, depending on which of the two choices was made. Going back to the above example, Figure 4.3 shows the 1-depth automaton A^1_w for the word $w = title.date.GetTemp.TimeOut$, with the signature of the $GetTemp$ and $TimeOut$ functions defined as in Section 4.1. q_2 and q_3 are the fork nodes and their two outgoing edges represent their fork options for $GetTemp$ and $TimeOut$, resp. An ϵ edge represents the choice of invoking the function while a function edge represents the choice not to invoke it. Note that a node may be a fork node for several function calls (e.g. if A^k_w contains a choice such as $(GetTemp|TimeOut)$). To handle that, we keep track of the correspondence between ϵ edges and the associated function calls.

Suppose first that we want to verify that all possible rewritings lead to a "good" word, i.e. that they belong to the target language R. To put things in

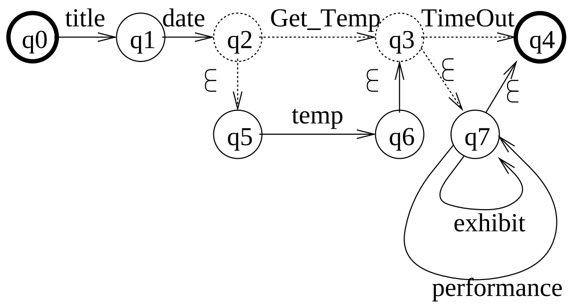

Figure 4.3: The A_w^1 automaton from the newspaper document.

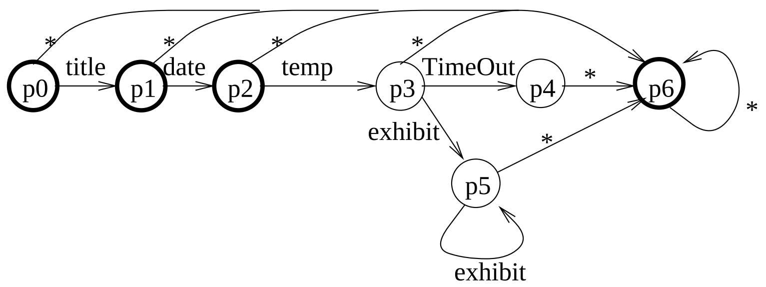

Figure 4.4: The complement automaton $\overline{A}$ for schema τ'.

regular language terms, the intersection of the language of A_w^k, consisting of these words, with the *complement* of the target language R should be empty. A standard way to test that the intersection of two regular languages is empty is to (i) construct an automaton $\overline{A}$ for the complement of the language R, (ii) build a Cartesian product automaton $A_\times = A_w^k \times \overline{A}$ for the two automata A_w^k and $\overline{A}$, and (iii) check whether it accepts no words.

The Cartesian product automaton of A_w^k and $\overline{A}$ is built in Step 11 of the algorithm. To continue with the above example, the complement automaton for the regular language $R = \tau'(newspaper)$ of the schema τ' is given in Figure 4.4. The accepting states are p_0, p_1, p_2 and p_6. For brevity we use "*" to denote all possible alphabet transitions *besides* those appearing in other outgoing edges. The Cartesian product automaton $A_\times = A_w^1 \times \overline{A}$ (where A_w^1 and $\overline{A}$ are the automata of Figures 4.3 and 4.4, resp.) is given in Figure 4.5. The initial state is $[q_0, p_0]$ and the final accepting one is $[q_4, p_6]$.

Note however that, when searching for a safe rewriting, one does not need to verify that all possible rewritings lead to a 'good word', i.e., that *none* of

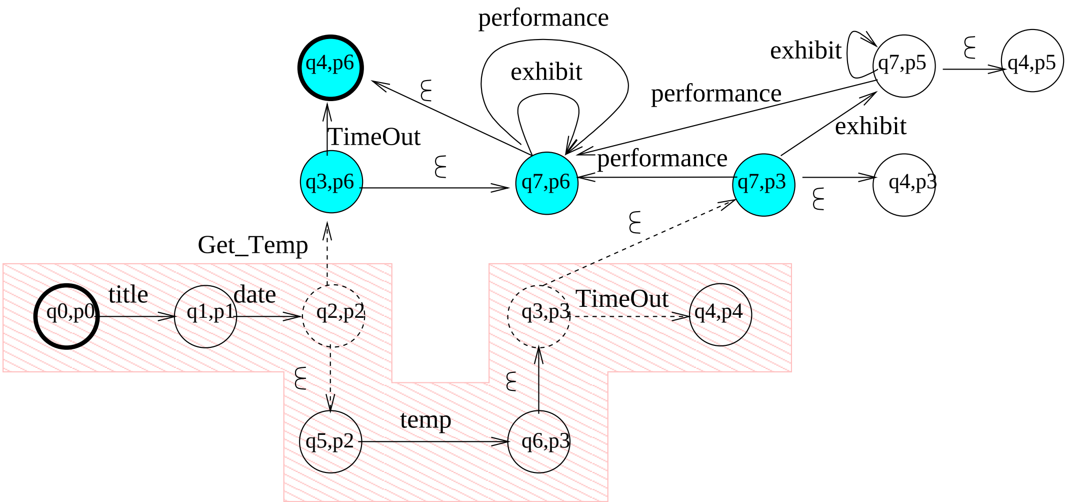

Figure 4.5: The Cartesian product automaton $A_\times$.

the words in A_w^k belongs to $\overline{A}$. We only have to verify that for each function, there is *some* fork option (i.e. invoking the function or not) that, if taken, will not lead to an accepting state. Since we are looking for left-to-right safe rewritings, we need to check that, traversing the input from left to right, at least one such 'good' fork options exists for each function call on the way. The marking of nodes in Steps 15-17 of the algorithm achieves just that. Recall that we required in Step (4) that the complement automaton $\overline{A}$ be *complete*. This is precisely what guarantees that all the fork nodes/options of A_w^k are recorded in $A_\times$ and makes the above marking possible.

The marking for our particular example is illustrated in Figure 4.5. The colored nodes are the marked ones. As can be seen, the fork nodes $[q_2, p_2]$ and $[q_3, p_3]$ are not marked. For the first node, this is because its ϵ fork option is not marked. For the second one, it is due to the unmarked *TimeOut* fork option. Consequently, the initial state is not marked as well and there is a safe rewriting of the newspaper element to the schema τ'. We will see in Section 4.3.4 how to find this rewriting.

For another example, consider the schema τ''. Here, a *newspaper* is required to have the structure conforming to the regular expression *title.date.temp.exhibit**. The complement automaton $\overline{A'}$ for this language is given in Figure 4.6. To test whether it is possible to safely rewrite our

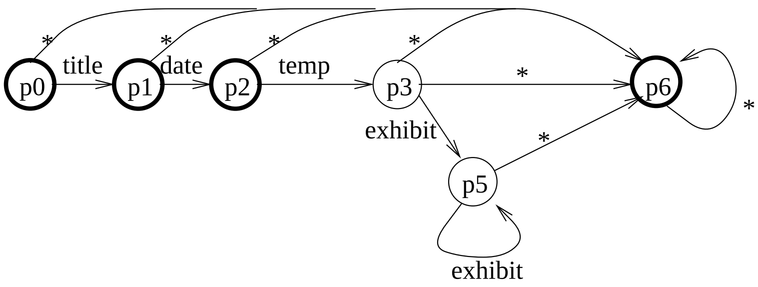

Figure 4.6: The complement automaton $\overline{A'}$ for schema τ''.

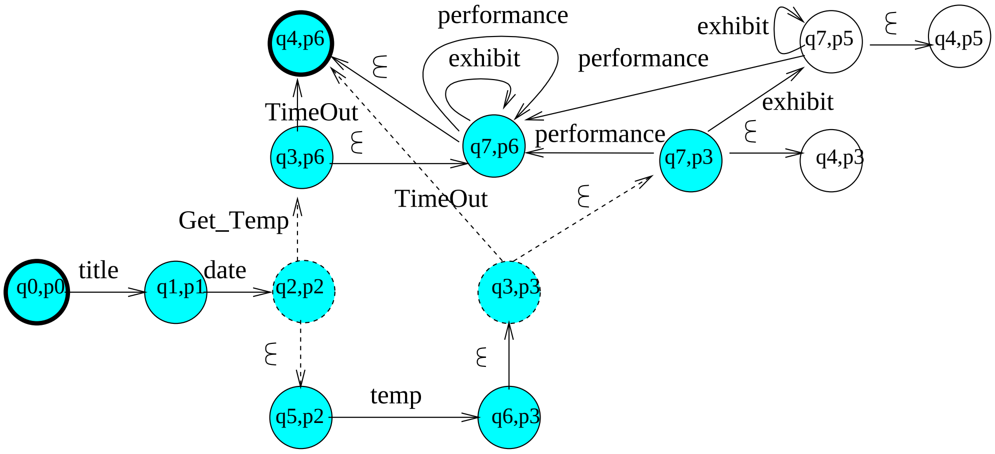

Figure 4.7: The Cartesian product automaton $A'_\times$.

newspaper document into this schema, we construct a Cartesian product automaton $A'_\times = A^1_w \times \overline{A'}$ (with A^1_w as in Figure 4.3 and $\overline{A'}$ as in Figure 4.6). $A'_\times$ is given in Figure 4.7.

As one can see, in this case, the two fork nodes $[q_2, p_2]$ and $[q_3, p_3]$ have both their fork options marked. Consequently the initial state is marked as well and there is no safe rewriting of w into the schema τ''. Note that this is precisely what our intuitive discussion from Section 5.1 indicated: the invocation of *TimeOut* may return *performance* elements, hence the result may not conform to the desired structure.

The following theorem states the correctness of our algorithm.

Theorem 4.3.1 *The above algorithm returns true if and only if a k-depth left-to-right safe rewriting exists.*

Proof: To prove correctness we have to show that (i) when the algorithm returns a negative answer a safe rewriting indeed does not exist, and (ii) when the answer is positive, there exists a safe rewriting.

Notations We will use the following notations:

- For an automaton X and a state $q \in X$, we denote $L(X, q)$ the language accepted by X when making q the initial state. This is a subset of all suffixes of words accepted by the original automaton X.
- In the automaton $A_\times$, we use A^0 to denote the sub-automaton "originating" from A_w, and use A^j, $0 < j \leq k$ to denote the sub-automaton "originating" from some A_f added to A_k to represent the possible outputs of f, at the j^{th} iteration of its construction. More formally, these are the projections of $A_\times$ on nodes $[q, q']$ such that q belongs to A_w for A^0, and to A_f for A^j. Note that in general, several automata are added in the j^{th} ^ iteration. To simplify the notation we use A^j to denote any representative of this set.
- Given a sub-automaton A^j, an initial (resp. final) state of A^j is a state $[q, q']$ such that q is an initial (resp. final) state of A_w/A_{f_i}.

Completeness We start by proving that the algorithm is complete, i.e. that if it answers negatively, no k-depth left-to-right safe rewriting of A_w exists.

We first number the nodes of $A_\times$ based on the order in which they got marked by the algorithm. For a regular node, its assigned number should be greater than the one of its marked successor that caused its marking. For a fork node the assigned number should be greater than all the numbers of the nodes that cause its marking. It is easy to come up with such a numbering by following the algorithm and using a counter that is incremented by one each time a node gets marked.

We also need the following lemma.

Lemma 4.3.2 *If a state $[q, q']$ of A^j is marked, then there exists a finite path from $[q, q']$ to a marked final state $[p, p']$ of A^j, such that all the nodes on the path belong to A^j, are marked, and have decreasing numbers. Moreover, either $[p, p']$ is a final state of A^0, or there is an outgoing ϵ edge from it to a marked state of some A^{j-1}, with a smaller number.*

Proof: First, by the definition of marking, there exists a *finite* marked path from $[q, q']$ to a final state of $A_\times$, where the nodes have decreasing numbers. If $[q, q']$ is in A^0, note that the final state of $A_\times$ is also a final state of A^0. Otherwise, by construction of $A_\times$, such a path must go through a marked final state of A^j and continue, via an ϵ edge to a marked state in A^{j-1} with a smaller number.

Among such paths, let us look at one that has a maximal marked prefix in A^j [6]. We denote $[p, p']$ the last node of the prefix. If $[p, p']$ is a final state of A^j that exits via an ϵ edge to a marked A^{j-1} node with a smaller number, we are done. Let us suppose it is not.

If $[p, p']$ is a fork node, then it has at least two outgoing edges that lead to marked states: a function transition, which stays in A^j, and the corresponding ϵ transition, which leads to some A^{j+1}. By definition of our numbering, both successor nodes have smaller numbers than $[p, p']$. Thus, we can extend our prefix in A^j by following the function transition, which contradicts the fact that we were on the path with a maximal prefix in A^j .

If $[p, p']$ is not a fork node, then it must have a marked successor with a smaller number (as otherwise it would not be marked). Its successors can either be in A^j or be ϵ transitions to some A^{j-1} (if it is a final state of A^j). As we assumed above that the ϵ transitions did not lead to marked nodes with lower number, $[p, p']$ must have a marked successor in A^j with a lower number, that caused its marking. Note however that by adding this marked node to the previous prefix, we can build a marked path with a longer prefix in A^j, having nodes with decreasing numbers. Again, a contradiction.$\square$

We are now ready to prove (i). The proof is again by contradiction. Assume that the algorithm returns a negative answer, i.e. that the initial

[6] Note that it is not necessarily unique.

state of $A_\times$ is marked, but a k-depth left-to-right safe rewriting from A_w does exist. Recall that such rewritings discover the input word and the answers of functions from left to right, and make their decisions (namely, to invoke functions or not) as they proceed. Therefore, we can construct the counter example incrementally. We do not need to provide the full input word (or the functions output) as a whole, but only "letter by letter", as the rewriting process is going on.

Also recall that, since the rewriting is supposed to be safe, we are free to choose any answer we want for a function call, as long as it matches its output type. The rewriting should succeed anyway.

We will show that we can provide a finite sequence of letters (consisting of an initial word and answers to the function calls the rewriting decides to invoke) that stays on a marked path in $A_\times$ and eventually reaches one of its final states. This means, on the one hand, that the sequence represents a legal k-depth rewriting (of a word accepted by A_w) and, on the other hand, that it does not belong to the target type. Consequently the rewriting is not safe.

The sequence is constructed as follows. We begin at the initial (marked) state of $A_\times$ and start following some finite marked path in A^0 leading to a marked final state where the nodes on the paths have decreasing numbers. Such a path must exist, by Lemma 4.3.2.

At each step, when we traverse an edge with a label in $\mathcal{L}$ we simply output its label. If the edge is labeled by a function name in $\mathcal{F}$, we also output the label, but our action depends on whether the rewriting process decides to invoke the function or not: If the function is not invoked, we stay on the same path. Otherwise, we follow an ϵ edge from the current automaton A^i (initially $i = 0$) to a marked state of the next level automaton A^{i+1}. Note that such a marked state must exist since the previous fork node was marked. Also, by definition of the numbering, its assigned number is smaller than the one of the fork node.

Then we continue the same process at A^{i+1} following a finite path, with nodes having decreasing numbers, to its final state, and on the way possibly moving to higher level automata, as described above.

Since all these paths are constructed as in Lemma 4.3.2, they end on a final node of $A_\times$ (for A^0), or on a final node of A^j with an ϵ transition to a marked node of A^{j-1}, with a smaller number. In the latter case, we simply follow this transition, which corresponds to ending the answer of a function call.

Observe, that by the above arguments, we follow a path to a final state of $A_\times$ that consists only of marked nodes and correspond to a decreasing sequence of numbers, which means that it is finite. Feeding the letters on this path to the safe rewriting make it end on a word that is not in the target language R, a contradiction.

Soundness We now turn to (ii), and first prove that if the algorithm succeeds, then the following proposition holds:

Proposition 4.3.3 *Let A^j be a sub-automaton of $A_\times$ corresponding to some function automaton A_{f_i} (or to A_w if $j = 0$).*

For every non marked state $[q, q']$ of A^j originating from A_{f_i} (resp. A_0), every word in $L(A_{f_i}, q)$ (resp. $L(A_w, q)$) has a "safe rewriting" into a word w' such that w' corresponds to a non marked path in $A_\times$ leading to a (non-marked) final state of A^j [7].

Proof: The proof is by induction on j, starting from $j = k$ and going down to $j = 0$, to show that every word in $L(A_{f_i}, q)$ (resp. $L(A_w, q)$) that contains only function nodes of depth $\geq j$ can be safely rewritten. For $j = k$, A^j contains no fork nodes. A state $[q, q']$ is thus not marked iff all states reachable from it are not marked and the property trivially holds.

We suppose now that the hypothesis holds for $j + 1$, and consider a word that contains function nodes of depth $\geq j$. We follow its corresponding path in A^j. If all nodes are non marked, we get to a non-marked accepting state of A^j, which means we are done. Otherwise, if the path contains marked

[7] We overload here, in a natural manner, the notion of safe rewriting, meaning that the above property holds no matter what answer the function invocations return.

nodes, we show, by a second induction on the number of function symbols in w, how to rewrite w to a "good" path.

The basis of the induction is for a word w that does not contain function nodes. Then, clearly, all nodes on the corresponding path must be non marked, or else the first one, $[q, q']$, would be marked as well. Suppose we know how to deal with a word containing l function nodes, and consider a word w that contains $l + 1$ function nodes. We look at the first edge $e = ([v, v'], [u, u'])$ on the path where $[v, v']$ is not marked but $[u, u']$ is marked. By definition of marking, $[v, v']$ must be a fork node with e labeled by some function name f_i. This splits w into sub-words $w = w_1.f_i.w_2$. Since $[v, v']$ is not marked, its other fork option (the ϵ edge corresponding to f_i), must lead to a non marked initial state of the sub-automaton A^{j+1} corresponding to A_{f_i}. We *choose* to invoke this function. By the first induction hypothesis, there is a safe rewriting of the returned result into a word w' whose corresponding path is not marked and leads to a non marked final state of A^{j+1}. By the construction of $A_\times$ this final state must have an outgoing ϵ edge leading to a state $[u, u'']$ of A^j. Furthermore, observe that the latter is not marked (or otherwise the final state of A^{j+1} would be also marked).

Finally, since w_2 has l function nodes, by the second induction hypothesis it can be safely rewritten into some word w'' whose corresponding non marked path leads to a final state of A^j. It follows that the rewritten word $w_1.w'.w''$, and its corresponding non marked path, leads from $[q, q']$ to a non marked final state A^j via a path consisting only of non marked nodes.$\Box$

We are now ready to prove (ii). If the algorithm answers positively, a safe rewriting can be found by essentially the same construction as of the above proposition.

Given any word w accepted by A_w, our goal is to find a safe rewriting that yields a word w' whose corresponding path in $A_\times$ leads to a non marked state $[q, p]$, where q is an accepting state of A_w (namely an accepting state of A^0). Note that since the final state is not marked, p is not an accepting state of $\overline{A}$. And since $\overline{A}$ is deterministic this implies that w' is a "good" word that belongs to the target language R.

The fact that such a rewriting indeed exists follows immediately from the above proposition, taking the node $[q, q']$ of the proposition to be the initial state of $A_\times$, and w as a particular input word. The actual rewriting can be found as described in the proof. This concludes the proof of Theorem 4.3.1.□

Complexity We now briefly discuss the complexity of the algorithm. Recall that we use s_0 to denote the schema of the sender and s to denote the agreed data exchange schema. The complexity of deciding whether a safe rewriting exists is determined by the size of the Cartesian product automaton: we need to construct it and then traverse and mark its nodes. More precisely, the complexity is bounded by $O(\mid A_\times \mid^2) = O((\mid A_w^k \mid \times \mid \overline{A} \mid)^2)$. The size of A_w^k is at most $O((\mid s_0 \mid + \mid w \mid)^k)$ and the size of the complement automaton $\overline{A}$ is at most exponential in the automaton being complemented [53], namely at most exponential in the size of the target schema s. This exponential blow up may happen however only when s uses non deterministic regular expressions (i.e. regular expressions whose corresponding finite state automaton is non deterministic). Note however that XML Schema enforces the usage of deterministic regular expressions. Hence, for most practical cases, the complexity is polynomial in the size of the schemas s_0 and s (with the exponent determined by k).

4.3.4 Finding a rewriting

The algorithm of Figure 4.2 checks if a safe rewriting exists. The constructive proof we used to show its soundness entails a means of finding a rewriting sequence when a safe rewriting exists, that corresponds to the algorithm of Figure 4.8.

This algorithm finds the safe rewriting sequence by following a non-marked path. Each fork node on the path, together with its non-marked fork option, determines what needs to be done with the corresponding function - an ϵ edge means “invoke the function” while a function edge means “do not invoke”. In the example previously described, which corresponds to Figure

find safe rewriting (word w, marked automaton $A_\times$)
To obtain such a rewriting:
(a) Follow a non marked path (corresponding to w) starting from the initial state of $A_\times$ to a state $[q, p]$ where q is an accepting state of A_w^k.
– The non marked *fork options* on the path determine the rewriting choices (i.e. which functions to call).
– When a function is invoked we continue the path with the new rewritten word (rather than the original w).
(b) To minimize the rewriting cost, choose a path with minimal number/cost of function invocations.
exit

Figure 4.8: Finding the rewriting of w into R.

4.5, it is easy to see (following the path with colored background) that $GetTemp$ needs to be invoked while $TimeOut$ should not.

The complexity of actually performing the rewriting depends on the size of the answers returned by the called functions. If x is the maximal answer size, the length of the generated word is bounded by $w \times x^k$.

4.3.5 A Mixed Approach

As seen above, much of the work in searching for a safe rewriting comes from the size of the automaton A_w^k that accounts for *all* possible outputs of function invocation. A useful heuristic is to adopt a mixed approach, that starts by invoking some of the functions (e.g. the ones with no side effects or low price) to get their actual output, and then tries to safely rewrite the document. In terms of the algorithm of Figure 4.2, rather than using the full function signature automaton A_{f_i}, we will use a smaller one, that describes just (the type of) the actual returned result. This may greatly simplify the resulting automaton A_w^k. Moreover, the output of the already invoked calls can be reused when performing the actual rewriting, instead of reissuing these calls.

	possible rewriting (word w, functions output types $R_{f_1} \dots R_{f_n}$, target language R)
1	Build finite state automata for the following regular languages:
2	(a) An automaton A_w^k as in Figure 4.2
3	(b) An automaton A accepting the regular language R.
4	Construct the cartesian product automaton $A_\times = A_w^k \times A$.
5	Mark all nodes in $A_\times$ having some outgoing path leading to a final state.
6	Return **true** if the initial state is marked, **false** otherwise.

Figure 4.9: Possible rewriting of w into R.

4.4 Possible Rewriting

We considered safe rewritings in the previous section. We now turn to possible rewritings. While function signatures provide an "upper bound" of the possible output, when invoked with the actual given parameters they may return a restricted "appropriate" output, so a rewriting that looked non feasible (unsafe) may turn to be possible after some function calls. To test if a rewriting *may* exist, we follow a similar three-step procedure as for safe rewriting: (1) test function parameters first, (2) traverse the tree top down, and (3) check each node individually, trying to rewrite the word w consisting of the labels of its direct children.

Steps (1) and (2) are exactly as before. For Step (3), Figure 4.9 provides an algorithm to test if the children of a given node *may* rewrite to the target schema. As before, we use the automaton A_w^k that describes all the words that may be derived from the word w in a k-depth rewriting. w *may* rewrite to a word in the target language R iff some of these derived words belong to R. Namely, if the intersection of the two languages, A_w^k and R, is not empty. To test this, we construct (in step 4 of the algorithm) the cartesian product automaton for these two languages, and test (in step 5) whether the final state is reachable from the initial one. This is done by a standard marking process, that starts from the final nodes, and marks all nodes that have some edge leading to a marked node. If the initial state is marked, this means that the intersection of the two languages is not empty [53].

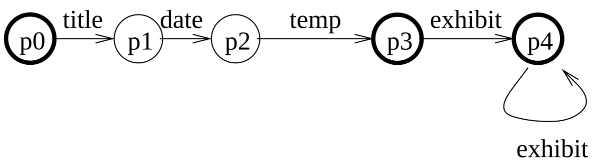

Figure 4.10: An automaton A for schema τ''.

For instance, consider the automaton A for the schema τ'' with *newspaper* having the structure *title.date.temp.exhibit** given in Figure 4.10. The initial state is p_0 and the final accepting states are p_3 and p_4. The cartesian product automaton $A_\times = A^1_w \times A$ (for A^1_k as in Figure 4.3 and A as in Figure 4.10) is given in Figure 4.11. The initial state is $[q_0, p_0]$. The final accepting states are $[q_4, p_3]$ and $[q_4, p_4]$, and all states (including the initial one) have an outgoing path to a final state. The only possible *fork options* left in the automaton, and which may lead to a possible rewriting, are the ones requiring the invocation of both *GetTemp* and *TimeOut* functions. If *TimeOut* returns nothing but *exhibits* the rewriting succeeds.

The correctness of this algorithm is stated below.

Proposition 4.4.1 *The above algorithm returns true iff a k-depth possible rewriting exists.*

Proof: Since A^k_w accepts the language of all possible words obtainable by a k-depth rewriting, the rewriting is possible iff the intersection of the language accepted by A^k_w with the target language is not empty. This is classically checked by computing the cross-product of the corresponding automata, and marking nodes as described, to checked whether a final state is reachable from the initial state.□

The complexity here is again determined by the size of the cartesian product automaton. However, in this case, it uses the schema automaton A (rather than its complement, as for safe rewriting). Hence, the complexity of checking whether a rewriting may exist is polynomial in the size of the schemas s_0 and s (with the exponent determined by k).

Finding an actual rewriting is done through a heuristic described by the algorithm of Figure 4.12. We follow a marked path, and invoke functions

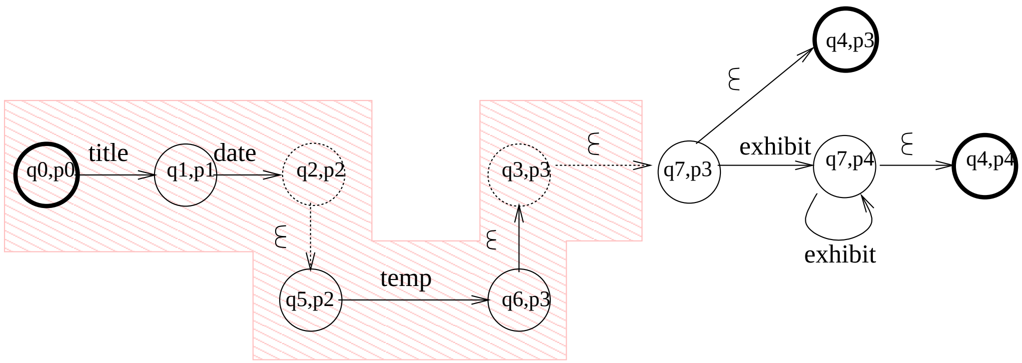

Figure 4.11: Cartesian product automaton for *possible* rewriting.

find possible rewriting (word w, marked automaton $A_\times$)

To obtain such a rewriting:
(a) Follow a marked path (corresponding to w) from the initial state of $A_\times$ to a final one, with the *fork options* on the path determining the rewriting choices (as in Figure 4.2).
(c) Backtrack when the calls return a value that does not allow to continue to an accepting state.
(d) To minimize the rewriting cost, chose a path with minimal number/cost of function invocations.
Exit when a final state is reached, or all choice were tried.

Figure 4.12: Finding a possible rewriting of w into R.

or not, as indicated by the *fork options* on the path. We have to backtrack when failing (i.e. when the function returns a value that does not correspond to an accepting path). This process ends either because we reached a final state, which means that a rewriting was found, or because all choices were explored without success.

4.5 Schema Rewriting

So far, we considered the rewriting of a single document. At a higher level, to check compatibility between applications, the sender may wish to verify that *all* documents generated by her application can indeed be sent to the target receiver. Given a schema s_0 for the sender documents, and some distinguished root label r, we want to verify that *all* instances of s_0 with root r can be safely rewritten to the schema s. Interestingly, it turns out that safe rewriting for schemas is not more difficult than for documents. We decompose the algorithm we propose for schema rewriting into two parts: First, how to check the initial schema, by traversing it top-down and second, for each type in this schema, how to check that the corresponding regular expression safely rewrites into the target schema.

We first show how safe rewriting can be checked for DTD's, by checking all the element definitions of s_0. Then, we sketch a top-down algorithm for checking safe rewriting for XML Schema-like schemas. Finally, we explain how it can be checked that any instance of a regular expression can be safely rewritten into a target regular expression.

4.5.1 Rewriting DTDs

In the simple DTD-like schemas we used so far, checking that s_0 safely rewrites to s amounts to checking that for every element definition $\tau_0(l_0) = r_0$ in s_0, (a) there exists an element definition for the element label l_0 in s and, (b) that every instance of the regular expression r_0 can be safely rewritten into the corresponding regular expression in s, namely $\tau(l_0)$. We term this last step *language safe rewriting*, and give an algorithm for it in Section 4.5.3.

Notice that for such simple schemas, the element definitions can be checked independently from each other, in any order. s_0 safely rewrites into s iff the language safe rewriting succeeds for all element definitions.

4.5.2 Rewriting XML Schemas

Things are more involved when we consider more expressive schema languages, in the style of XML Schema. Types are decoupled from element labels, but the type of an element is unambiguously determined by its label and the type of its parent. In this case, schema rewriting can be checked by a top-down analysis of the initial schema s_0, starting from the root. The type of the root determines the regular expression that has to be matched by its children, and the type of the root of s determines the target regular expression for the safe rewriting of types.

Then, recursively moving down, the types corresponding to the labels of the children on both sides are unambiguously determined, and so are their corresponding regular expressions. Therefore safe rewriting of types can be checked at the next level, and so on.

Notice that, while proceeding this way, only pairs of types for which safe rewriting has not been tested yet need to be processed. This ensures that the algorithm terminates, even if schemas are recursive.

4.5.3 Language safe rewriting

We explain now how to check for language safe rewriting. Given two regular expressions R_0 and R, we want to check that all words in the language of R_0 have a safe rewriting into a word in the language of R. The algorithm of Figure 4.13 checks just that.

This algorithm is almost identical to the one presented in Section 4.3, except that the initial automaton is built to accept the language R_0 instead of a single word. The following proposition states its correctness.

Proposition 4.5.1 *The above algorithm returns true if and only if every word in the language R_0 has a k-depth left-to-right safe rewriting into a word of R.*

Proof: The proof of the algorithm of Section 4.3 naturally extends to language safe rewriting. There, the completeness of the algorithm was shown by building a counter-example to the fact that there might be a safe rewriting although the algorithm answers negatively. The same construction holds for language rewriting, since it suffices to show that one word in the language R_0 does not safely rewrite into R to contradict the fact that R_0 does not rewrite into R. The soundness of the algorithm of Section 4.3 was shown in a constructive manner, by building a word corresponding to a non-marked path in $A_\times$. The same construction applies to *each* word accepted by A_{R_0},

language safe rewriting (initial language R_0,
functions output types $R_{f_1} \dots R_{f_n}$,
target language R)

Build finite state automata for the following regular languages:
(a) An automaton A_{R_0} accepting the regular language R_0.
(b) Automata A_{f_i}, $i = 1 \dots n$, respectively accepting the regular languages R_{f_i}.
(c) An automaton $\overline{A}$ accepting the complement of the regular language R. The automaton should be deterministic and *complete*, namely each state must have outgoing edges for all possible letters.
Let $A^k_{R_0} := A_{R_0}$.
For $j = 1, \dots, k$
Consider all the edges $e = (v, u)$ in $A^k_{R_0}$ that are labeled by an (invocable) function name f_i and were not treated in previous iterations. For each such edge:
(a) extend $A^k_{R_0}$ by attaching a copy of the automaton A_{f_i}, with its initial and final accepting states linked to v and u resp. by ϵ moves.
(b) Denote v as a *fork* node (for the edge e).
(c) The two *fork options* of v (for e) are e itself and the new outgoing ϵ edge.
Construct the cartesian product automaton $A_\times = A^k_{R_0} \times \overline{A}$.
The *fork nodes* and *fork options* in $A_\times$ reflect those of $A^k_{R_0}$:
(a) the *fork nodes* $[q, p] \in A_\times$ are those where q was a fork node in $A^k_{R_0}$.
(b) Similarly, a *fork option* in $A_\times$ consists of all edges originating from one fork option edge in $A^k_{R_0}$.
Mark nodes in $A_\times$ as follows.
(a) First mark all accepting states, (namely nodes $[q, p]$ where q and p are accepting states in $A^k_{R_0}$ and $\overline{A}$ resp.
(b) Then iteratively: mark regular (non fork) nodes if one of their outgoing edges points to a marked node; mark fork nodes if in both their fork options (for some f_i) contain an edge that points to a marked node.
Return **true** if the initial state is marked, **false** otherwise.

Figure 4.13: Language safe rewriting of R_0 into R.

i.e. for each word in the language R_0, which establishes the correctness of this algorithm.□

4.6 Implementation aspects

The ideas and algorithms presented in the previous sections were implemented as a pluggable *Schema Enforcement* module of our system. Next, we present how the abstract schema language used in this chapter maps to XML Schema, SOAP and WSDL. Then, we briefly describe the *Schema Enforcement* module.

4.6.1 Using the standards

In order to define schemas for AXML documents, we use XML Schema$_{int}$, which is an extension of the XML Schema language we introduced in Chapter 2. To describe intensional data, XML Schema$_{int}$ introduces *functions* and *function patterns*. These are declared and used like element definitions in the standard XML Schema language. In particular, it is possible to declare functions and function patterns globally, and reference them inside complex type definitions (e.g. sequence, choice, all). We give next the XML representation of *function patterns* that are described by a combination of five optional attributes and two optional sub-elements: *params* and *return*:

```
<functionPattern
    id = NCName              methodName = token
    endpointURL = anyURI     namespaceURI = anyURI
    WSDLSignature = anyURI   ref = NCName>
  Contents: (params?, return?)
</functionPattern>
```

The *id* attribute identifies the function pattern, which can then be referenced by another function pattern using the *ref* attribute. Attributes *methodName*, *endpointURL* and *namespaceURI* designate the SOAP Web service that implements the boolean predicate used to check whether a particular function matches the function pattern. It takes as input parameter the SOAP identifiers of the function to validate. As a convention, when these parameters are omitted, the predicate returns `true` for all functions. The *Contents* detail the function signature, i.e. the expected types for the input parameters and the result of the function. These types are also defined using XML Schema$_{int}$, and may contain intensional parts.

To illustrate this syntax, consider the function pattern *Forecast*, that captures any function with one input parameter of element type *city*, returning an element of type *temp*. It is simply described by:

```
<functionPattern id="Forecast">
  <params>
    <param> <element ref="city"/> </param>
  </params>
  <result> <element ref="temp"/> </result>
</functionPattern>
```

Functions are declared in a similar way to function patterns, by using elements of type `function`. The main difference is that the three attributes `methodName`, `endpointURL` and `namespaceURI` directly identify the function that can be used.

As mentioned already, function and function pattern declarations may be used at any place where regular element and type declarations are allowed. For example, a *newspaper* element with structure

$$\tau(newspaper) = title.date.(Forecast|temp).(TimeOut|exhibit^*)$$

may be defined in XML Schema$_{int}$ as:

```
<xsd:element name="newspaper">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="title"/>
      <xsd:element ref="date"/>
      <xsd:choice>
```

```
        <xsi:functionPattern ref="Forecast"/>
        <xsd:element ref="temp"/>
      <xsd:/choice>
      <xsd:choice>
        <xsi:functionPattern ref="TimeOut"/>
        <xsd:element ref="exhibit" minOccurs="0"
                 maxOccurs="unbounded"/>
      <xsd:/choice>
  <xsd:/complexType>
<xsd:/element>
```

Note that, just as for documents, we use a different namespace (embodied here by the use of the prefix `xsi`) to differentiate the intensional part of the schema from the rest of the declarations.

Similarly to XML Schema, we require definitions to be unambiguous [110]. Namely, when parsing a document, for each element and each function node, the sub-elements can be sequentially assigned a corresponding type/function pattern in a deterministic way by looking only at the element/function name.

One of the major features of the WSDL language is to describe the input and output types of Web services functions using XML Schema. We extend WSDL in the obvious way, by simply allowing these types to describe intensional data, using XML Schema$_{int}$. Finally, XML Schema$_{int}$ allows WSDL or WSDL$_{int}$ descriptions to be referenced in the definition of a function or function pattern, instead of defining the signature explicitly (using the `WSDLSignature` attribute).

4.6.2 The Schema Enforcement module

To implement this module, we needed a parser of XML Schema$_{int}$. We had the choice between extending an existing XML Schema parser based on DOM level 3 or developing an implementation from scratch [79]. Whereas the first solution seems preferable, we followed the second one because, at the time we started the implementation, the available (free) software we tried (Apache Xerces [106] and Oracle Schema Processor [80]) appeared to have limited extensibility. Our parser relies on a standard event-based SAX parser [106]. It does not cover all the features of XML Schema, but implements the important ones such as complex types, element/type references and schema import. It does not check the validity of all simple types, nor does it deal with inheritance or keys. However, these features could be added rather easily to our code.

The schema enforcement algorithm we implemented in the module follows the main lines of the algorithm in Section 4.3, and in particular the three same stages:

1. Checking function parameters recursively, starting from the innermost ones and going out,
2. Traversing, in each iteration, the tree top down, and
3. Rewriting the children of every node encountered in this traversal.

Steps (1) and (2) are done as described in Section 4.3. For Step (2), recall from above that XML Schema$_{int}$ are deterministic. This is precisely what enables the top down traversal since the possible type of elements/functions can be determined locally. For Step (3), our implementation uses an efficient variant of the algorithm of Section 4.3. While the latter starts by constructing all the required automata and only then analyzes the resulting graph, our implementation builds the automaton $A_{\times}$ in a lazy manner, starting from the initial state, and constructing only the needed parts. The construction is pruned whenever a node can be marked directly, without looking at the remaining, unexplored, branches. The two main ideas that guide this process are the following:

- **Sink nodes** Some accepting states in $\overline{A}$ are "sink" nodes: once you get there, you cannot get out (e.g. p_6 in Figures 4.4 and 4.6). For the Cartesian product automaton $A_{\times}$, this means that *all* paths starting from such nodes are marked. When such a node is reached in the construction of $A_{\times}$, we can immediately mark it and prune all its outgoing branches. E.g., in Figure 4.14, the top left shaded area illustrates

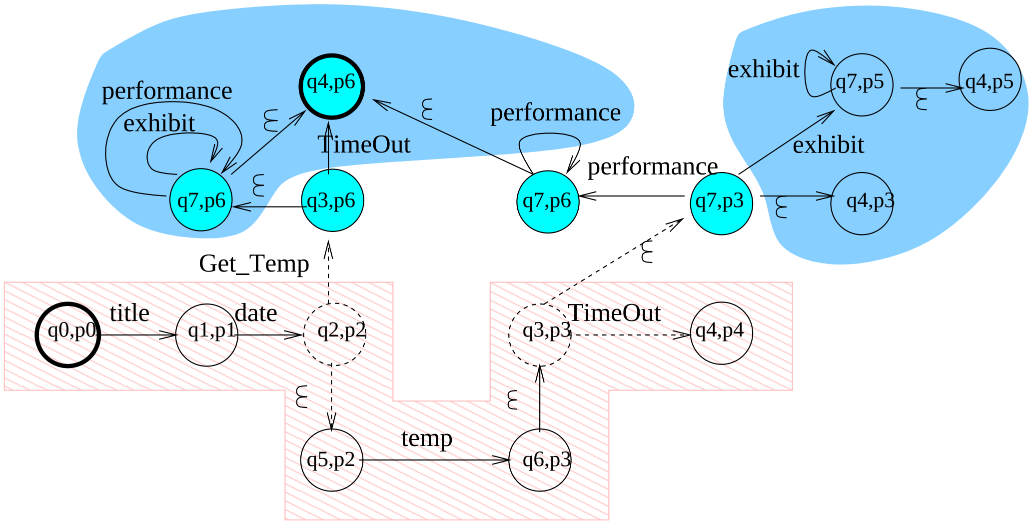

Figure 4.14: The pruned automaton.

which parts of the Cartesian product automaton of Figure 4.5 can be pruned. Nodes $[q_3, p_6]$ and $[q_7, p_6]$ contain the sink node p_6. They can be immediately be declared as marked, and the rest of the construction (the left shaded area) need not be constructed.

- **Marked nodes** Once a node is known to be marked, there is no point in exploring its outgoing branches any further. To continue with the above example, once the node $[q_7, p_6]$ gets marked, so does $[q_7, p_3]$ that points to it. Hence, there is no need to explore the other outgoing branches of $[q_7, p_3]$ (the shaded area on the right).

While this dynamic variant of the algorithm has the same worst-case complexity as the algorithm of Figure 4.2, it saves a lot of unnecessary computation in practice. Details are available at [79].

4.7 Conclusion

As mentioned in the previous chapter, the idea of XML documents with embedded calls to Web services (or close variants to this paradigm) are already present in several existing systems: relational and object databases, and scripting languages for the Web (e.g. JSP [62]). While the static structure of such documents can be described by some DTD or XML Schema, our extension of XML Schema with function types is a first step towards a more precise description of XML documents embedding computation. Further work in that direction is clearly needed to better understand this powerful paradigm.

In many applications, it is necessary to screen queries and/or results according to specific user groups [28]. More specifically for us, embedded Web service calls in documents that are exchanged may be a serious cause of security violation. Indeed, this was one of the original motivations for the work presented here. Controlling these calls by enforcing schemas for exchanged documents appeared to us as useful for building secure applications, and can be combined with other security and access models that were proposed for XML and Web services, e.g. in [37, 68]. However, further work is needed to investigate this aspect.

We presented in this chapter what forms the core of the module that, in our system, supports and controls the dialogue (via Web services) with the rest of the world. This particular module may be extended in several ways. First, one may introduce "automatic converters" capable of restructuring the data that is received to the format that was expected, and similarly for the data that is sent. Also, this module may be extended to act as a "negotiator" who could speak to other systems to agree with them on the intensional XML Schemas that should be used to exchange data. Finally, the module may be extended to include search capabilities, e.g., UDDI style search [97], to try to find services on the Web that provide some particular information.

Note that when some data may be found in different places and a service may be performed at different sites, the choice of which data to use and where to perform the service becomes an optimization issue. This is related to work on distributed database systems [94] and to distributed computing at large. The novel aspect here is the ability to exchange intensional information. This is in spirit of [61], that considers also the exchange of intensional information in a distributed query processing setting.

As mentioned already in the previous chapter, Intensional XML documents nicely fit in the context of data integration, since an intensional part of an XML document may be seen as a view on some data source. Calls to Web services in XML data may be used to wrap Web sources [43] or to propagate changes for warehouse maintenance [117]. Note that the control of whether to materialize data or not (studied here) provides some flexible form of integration, that is a hybrid of the warehouse model (all is materialized) and the mediator model (nothing is). On the other hand, this is orthogonal to the issue of selecting the views to materialize in a warehouse, studied in, e.g., [49, 116].

To conclude this chapter, we mention some fundamental aspects of the problem we studied. Although the k-depth/left-to-right restriction is not limiting in practice and the algorithm we implemented is fast enough, it would be interesting to understand the complexity and decidability barriers of (variants of) the problem. As we mentioned already, inspired by our work, [76] studies some fundamental questions that are raised. They proved the undecidability of the general safe rewriting problem for a context-free target language, and provided tight complexity bounds for several restricted cases.

We already mentioned the connection to type theory and the novelty of our work in that setting, coming from the regular expressions in XML Schemas. Typing issues in XML Schema have recently motivated a number of interesting works such as [72], that are based on tree automata.

Chapter 5

Querying AXML data

We saw in the previous chapter how materialization could be guided by a schema, when an AXML document needs to be exchanged between two systems. We now turn to another situation where materialization of intensional data plays a crucial role, namely when a query needs to be answered on an AXML document.

To illustrate this notion, consider a Web site about your city's nightlife [1], which is described by an AXML document, containing information about, say, movies and restaurants. Now, suppose someone asks the query:

`/goingOut/movies//show[title= "The Hours"]/schedule`.

Then, there is no point in invoking any calls found below the path `/goingOut/restaurants`, since the data they return would be set in a place where they cannot contribute to the result. One would also like to avoid invoking a call found below `/goingOut/movies` with a signature indicating that the returned data is irrelevant to the query, e.g., a service calls returning movie reviews.

The previous observation rules out a naive approach that would consist in invoking *all* the calls in the document recursively, until a fixpoint is reached, and finally running the query over the resulting document. The query processing needs to be more sophisticated and in particular to call services selectively. An alternative approach would be to build a query processor that

[1] In the style of http://www.timeout.com

would traverse the document top-down, and invoke calls encountered while evaluating the query. This approach would materialize only part of the document – the subtree traversed by the top-down evaluation. However, the mixing of query processing and service invocation would result in bad performances, since the query processor would either have to be blocked waiting for call responses, or would have to be restarted several times to account for the document growth. Moreover, forcing a particular query processing strategy eliminates most opportunities for query optimization. Instead, the technique we propose consists in identifying in advance a tight superset of the service calls that should actually be invoked to answer a query. The idea is to first invoke these calls and then move to a classical query processing as a second stage.

Observe that while the problem we address is close to the classical mediation paradigm of data integration, where data sources are called whenever queries are asked on a mediated schema, things are substantially different in AXML. Service calls may appear (i) *anywhere* in the *data*, and (ii) *dynamically* in results of previously invoked calls. Moreover, the relevance of one call may *depend* on the result of another one, and the service calls *return types* should be taken into consideration. A first contribution we make in this chapter is a better understanding of the possible relationships among the service calls embedded in a document, and their influence on the relevance of calls to queries. Based on that, our central contribution here is an efficient algorithm for lazy query evaluation, with the following features:

- **Computing the set of relevant service calls.** Given a query, the algorithm generates a set of auxiliary queries that, when evaluated on a document, retrieve all service calls that are relevant to the query.

- **Service calls sequencing.** The relationships among the calls are analyzed to derive a sequence of call invocations appropriate to answer the query.

- **Pruning via typing.** The return types of services are used to rule out more irrelevant service calls.

- **Service calls guide.** A specialized access structure is used to speed up obtaining the relevant calls.

- **Pushing queries.** Precise knowledge of the interaction between the query and each service call enables pushing queries to Web services when possible, just like mediators do with their data sources.

The algorithm is *dynamic*, in the sense that it adapts to the state of the document, as this state is modified by service invocations, and decides at each point which services should be invoked next. Our experimental results, presented in Section 5.7, show that, compared to the naive approach, the pruning of irrelevant service calls may reduce the overall query evaluation time by orders of magnitude. They also demonstrate the gain obtained (more classically) from pushing queries to service providers.

We now mention the main differences between this and the work presented in the previous chapter. There, the goal was to identify the calls that need to be invoked in order to make a document match an XML schema, and the solution relied on schema analysis via automata-based algorithms. By contrast, our goal here is to find the sequence of call invocations needed to evaluate a query over AXML data, and the solution is based on query analysis and rewriting. Indeed the two techniques are complementary: The technique of the previous chapter can be used to ensure that the parameters of calls invoked in our query processing match the input schema required by the signature of the service, and can also be applied to query results if they need to be exchanged. Conversely, the technique considered here may be used to evaluate queries on exchanged AXML data.

This chapter is organized as follows: Section 5.1 illustrates the problem on a sample AXML document, introduces the query language, and formalizes the general problem. Section 5.2 explains how to find, for a given state of a document, the service calls that are currently relevant for a query. Then, using a delicate analysis of the relationship among the calls, we provide in Section 5.3 an efficient algorithm to derive the sequence of call invocations needed to answer a query. A refinement, based on the analysis of service signatures is described in Section 5.4. Two techniques to further speed up the computation, via query relaxation and a special access structure, are described in Section 5.5. The pushing of queries is considered in Section 5.6. The experimental setting and results are described in Section 5.7. The last section overviews works that are related to this specific problem and concludes.

5.1 Preliminaries

We start by describing a sample AXML document, which will be used to illustrate the techniques presented in this chapter. Then, to define the problem formally, we describe the query language we use, which captures the core tree-pattern matching fragment of XQuery. Finally, we introduce the new notion of *relevance* of service calls for queries, which will be central to the lazy query evaluation considered in further sections.

Sample document A sample AXML document is represented as a tree in Figure 5.1. Function nodes have bold labels, and are numbered so that they can be referred to in later discussions. Data values are quoted. This document contains a list of hotels, some of which are given explicitly and some only intensionally, through an embedded call to the `getHotels` function. The document details (extensionally or intensionally) for each hotel its name, address, rating, and some nearby restaurants or museums. When the first `getNearbyRestos` call is invoked with the address of the hotel as a parameter, it returns a list of `restaurant` elements, which replaces the function node, as shown in Figure 5.3 (ignore for now the gray marking). Recall that the parameter subtrees and the return values may themselves be AXML documents.

Signatures of Web services, namely the expected type of their parameters and results, are typically given in their WSDL description. We saw in the previous chapter that, for services with AXML input or output, these types also entail information about the intensional parts of the data, and detail which service calls may appear where. For simplicity, we use here the same DTD-like syntax as in Section 4.1. Signatures are described by a schema τ

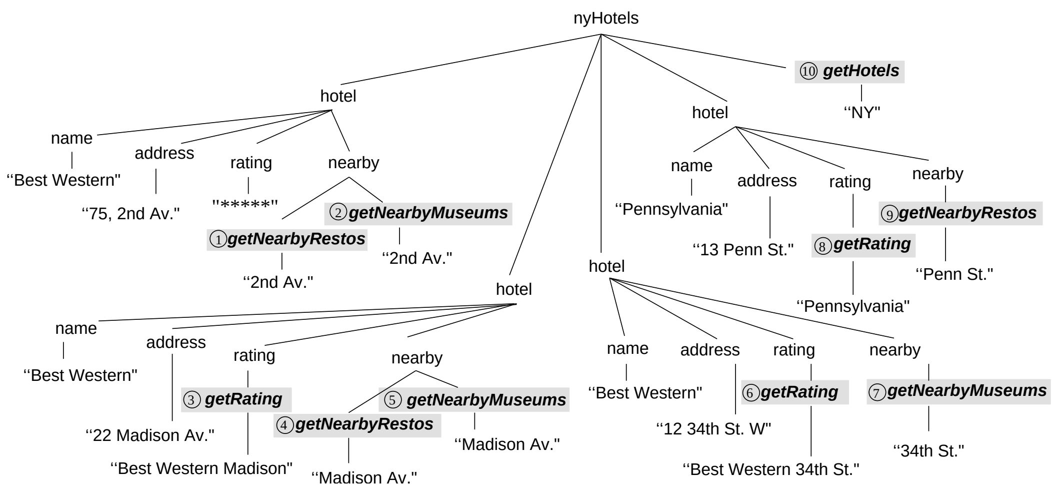

Figure 5.1: A sample Active XML document

that associates (1) a pair of regular expressions with each function name f, which respectively describe the input and output type of the function, and (2) a regular expression with each element name, which describes the structure of the element. The keyword `data` is used for data values. A sample schema τ can be seen in Figure 4.5.

Consider a particular function node. Its input must be properly typed, i.e. the labels of its children must match the corresponding τ_{in} regular expression. Its result is guaranteed to match the τ_{out} regular expression. Like in DTDs, for a data node occurring in an input/output subtree, the regular expression associated with its label must be matched by the labels of its children. It is easy to see that the output of `getNearbyRestos` in Figure 5.3 matches the corresponding output type in τ.

Queries Queries are modeled by *tree patterns*. A tree pattern query is a labeled tree whose nodes are labeled by variable names, constants (element names and data values), or the special symbol $*$. The nodes labeled by variable names are called *variable nodes* and those labeled by element names and data values are called *constant nodes*. The tree also has a distinguished set of edges called *descendant edges* and a distinguished set of nodes called the *result nodes*.

data:

$$
\begin{array}{lcl}
\tau(hotel) & = & name.address.rating.nearby \\
\tau(nearby) & = & restaurant^{*}.getNearbyRestos^{*} \\
 & & .museum^{*}.getNearbyMuseums^{*} \\
\tau(restaurant) & = & name.address.rating \\
\tau(museum) & = & name.address \\
\tau(name) & = & data \\
\tau(address) & = & data \\
\tau(rating) & = & (data \mid getRating)
\end{array}
$$

functions:

$$
\begin{array}{lcl}
\tau_{in}(getHotels) & = & data \\
\tau_{out}(getHotels) & = & hotel^{*} \\
 & & \\
\tau_{in}(getRating) & = & data \\
\tau_{out}(getRating) & = & data \\
 & & \\
\tau_{in}(getNearbyRestos) & = & data \\
\tau_{out}(getNearbyRestos) & = & restaurant^{*} \\
 & & \\
\tau_{in}(getNearbyMuseums) & = & data \\
\tau_{out}(getNearbyMuseums) & = & museum^{*}
\end{array}
$$

Figure 5.2: A schema τ for function signatures.

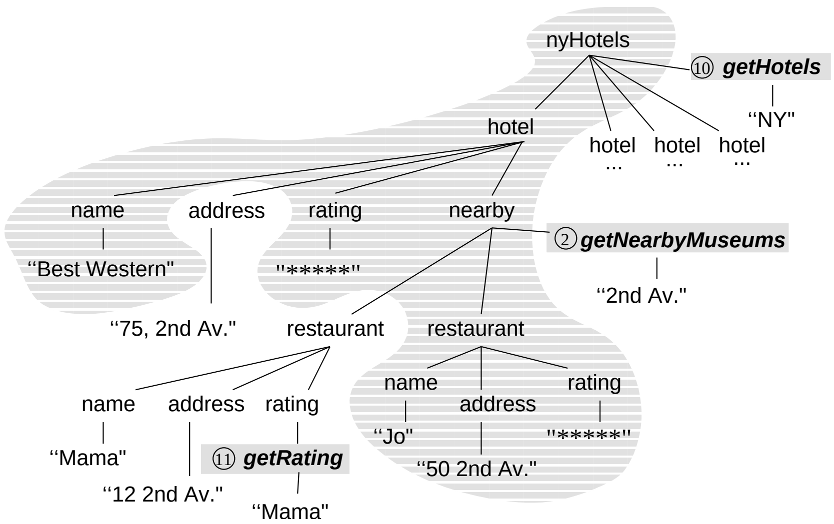

Figure 5.3: After invoking getNearbyRestos

Figure 5.4 shows a tree pattern query. Descendant edges are represented by doubled lines and output nodes are pointed by arrows.

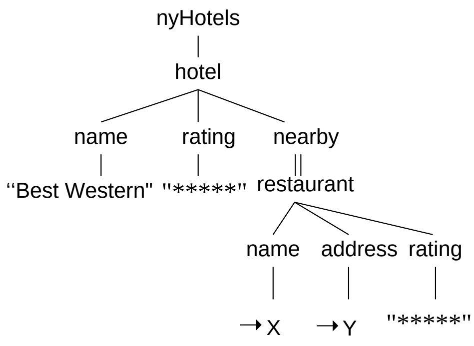

Figure 5.4: A tree pattern query

To define the semantics of queries, we use the notion of query embedding.

Definition 5.1.1 *Given a tree pattern query q and an AXML document d, an* embedding *of q into d is a tree homomorphism ρ from the nodes of q to the data nodes of d, mapping the root of q to that of d, preserving the parent-child and ancestor-descendant relationships, mapping each constant node of q to a data node of d with the same label, and such that all the variable nodes of q with the same variable name are mapped to data nodes having identical labels.*

The restriction of ρ to the result nodes of the pattern q is called the result *of the embedding. The set of results of all possible embeddings is called the* snapshot result *of the query, and denoted $q(d)$.*

Note that the data nodes of d that *contribute* to a given embedding are the images of the pattern nodes, plus the nodes in d residing on the paths between those images (i.e. d nodes that correspond to descendant edges in the pattern). Consequently, we say that a data node in d **contributes** to $q(d)$ if it contributes to some embedding of q in d.

For instance, the query of Figure 5.4 has no embedding in the document of Figure 5.1 and the snapshot result on this document is thus empty. On the other hand, once the `getNearbyRestos` function is invoked, the query has one embedding in the document of Figure 5.3. The marked grey area contains the nodes that contribute to the embedding, and the snapshot result on this document consists of the name and address of the marked restaurant.

Observe that, while AXML documents are ordered, our tree pattern queries are not sensitive to document order. This is because the former represent XML data (which is ordered), while the latter are intended to capture the core of XPath/XQuery, focusing on vertical axes like child and descendant, where the order of elements is irrelevant. A query language that would be order sensitive would require another analysis that we will not consider here.

It is also important to note that queries only match the *data nodes* of the document. This is because function nodes are only means to get the data they represent. Consequently, we distinguish between two possible semantics for a query: The *snapshot result*, defined above, is the result of the query when evaluated on the document *in its current state*, without invoking the function calls it contains. By contrast, the *full result* of a query represents (possibly intensionally) its snapshot result when evaluated on the document in its "full" state. Intuitively, by full state we mean the state in which all possible calls in the document have been evaluated, i.e., the document is fully materialized. The notions of full state and result are formalized in [5]. *In a nutshell, we focus here on efficiently computing the full result of a query, while intelligently invoking as few function calls as possible.*

Relevant calls When functions calls are invoked, their answers, inserted in the document, may contribute new embeddings to the query. As the document evolves, the snapshot result of the query may grow. We would like (i) to invoke only function calls that may indeed contribute to this growth, and (ii) to detect when no more function invocations can add significant data, and the full query result can be computed by simply evaluating the query on the document. These notions are formalized next.

Recall from the previous chapter that, for an AXML document d_1, $d_1 \xrightarrow{v} d_2$ if d_2 is obtained from d_1 by selecting a function node v in d_1 with some label f and replacing it by an arbitrary output instance of f. If $d_1 \xrightarrow{v_1} d_2 \xrightarrow{v_2} d_3 \ldots \xrightarrow{v_{n-1}} d_n$ we say that d_1 *rewrites* into d_n, denoted $d_1 \xrightarrow{*} d_n$. Also, recall that, in the rewriting process, the replacement of a function node v by an output instance is independent of any function semantics, and that we may replace calls to the same function by different output instances.

We say that a node $v \in d_n$ was *produced by* a function node v_i if $v \notin d_i$ and $v \in d_{i+1}$. v is *transitively produced by* v_i if it is either produced by v_i or is produced by some function node v_j transitively produced by v_i.

We are now ready to define the central notions of relevance and completeness.

Definition 5.1.2 *Given a document d, we say that a function node $v \in d$ is* relevant *for a query q if there exists some rewriting $d \xrightarrow{v} d_1 \xrightarrow{*} d_n$ where some of the data nodes transitively produced by v contribute to $q(d_n)$.*

If a document contains no relevant calls, it is said to be complete *for the query.*

Observe that the notion of function relevance has an *optimistic* nature: a function is considered relevant if it is possible that it returns a "good" result which, possibly together with additional "good" data produced by the invocation of other functions (in its own output or in other places in the document) will contribute to the query evaluation.

To illustrate that, consider the document of Figure 5.1 and the query of Figure 5.4. The relevant functions here are 1, 3, 4 and 10. Function 1 is relevant, as it may produce restaurant elements with a high rating, or a `getRating` function call that may produce such a rating, and hence make the hotel qualify for the query. Function 4 is relevant, as it may similarly produce highly rated restaurants, which, together with a potential high rating for the hotel, returned by function 3, may also make this hotel qualify. Function 3 is relevant for symmetric reasons. Function 10 may return qualifying hotels. Other function calls are irrelevant, either because their output is of a type that cannot contribute to the query (e.g. 7) or because, regardless of their answer, the corresponding hotel cannot satisfy the query criteria (e.g. 6 and 8).

Relevant rewritings Our goal is to invoke relevant calls in the document, until it is complete for the query, and then evaluate the query on it. An immediate issue that arises is *termination*: since function invocations may return new data and new function calls, a rewriting may never terminate. As we already mentioned, this behavior is inherent in the AXML model, and is carefully studied in [5], which provides sufficient conditions for termination. We will not consider this issue here, and assume that either these conditions hold or that the computation halts if a full state is not reached after some time limit. We focus here on the selective invocation of functions and completeness detection.

Note that once a function is invoked and returns a specific answer (one among all the possible instances of the function's output type), other functions that were considered relevant before may cease to be so. For instance, if Function 3 is invoked and returns a low rating, Function 4 is no longer relevant. This motivates the following definition.

Definition 5.1.3 *For a document d_1 and a query q, we say that a rewriting $d_1 \xrightarrow{v_1} d_2 \xrightarrow{v_2} d_3 \ldots \xrightarrow{v_{n-1}} d_n$ is* relevant *if $\forall i \in 1\ldots, n{-}1$, the function node v_i is relevant for d_i and q. The rewriting is* complete *if d_n is complete for q.*

In a relevant rewriting, only functions that may contribute to the query result are invoked. This is in contrast to the naive approach where all functions are invoked. Note however that there is a tradeoff between *accuracy* and *efficiency*: if it is expensive to exactly detect which calls are relevant and which are not, one may prefer a more lenient rewriting, where all relevant calls are indeed invoked but perhaps also some additional, non relevant ones. Note that such a rewriting is "safe", in the sense that it does not change the query result. The challenge is thus to find the right balance between the efforts spent on ruling out irrelevant calls and the actual time saved by avoiding their invocation.

Pushing queries Even when a call is relevant, its *entire* result may not be needed to evaluate the query. For instance, `getNearbyRestos` may return many restaurants. As we are only interested in five-star ones, and more precisely, only in their names and addresses, we may want to *push* to the function call a precise *subquery*, specifying that it has to apply the five-star rating selection, and only return the relevant names and addresses.

The results We can now state more formally the contributions of this chapter.

1. We present an algorithm that, given a query q and a document d, finds all the function calls in d that are relevant for q.

2. The above algorithm induces a simple mechanism to find a relevant rewriting, that iteratively invokes relevant calls, and applies the previous algorithm at each step on the changed document. We improve on this, by analyzing dependencies among relevant functions, and minimizing the needed efforts to find, at each step of the rewriting, the next function to be invoked.

3. We analyze the complexity of the algorithm, and propose two complementary methods to improve performance. The first uses a lenient variant of the algorithms, with a less refined but more efficient call relevance detection component. The second is based on an access structure to speed up processing.

4. We provide an algorithm that determines the subqueries to be pushed to the function calls that we decide to trigger, and a simple execution model for controlling this distributed computation.

5. Finally, we provide an experimental assessment of the performances of the above techniques.

Some useful machinery Before presenting our results, we need to introduce two extensions to the query language, that will be useful in the sequel. We will use the following two new types of nodes in tree pattern queries.

The first are *`OR`-nodes*, that represent a *choice* between their children subtrees. A query q with `OR`-nodes can be viewed as the union of all queries

q_i without `OR`-nodes that can be obtained from q by some choice of a single child subtree for each `OR`-node in q.

The second are *function nodes*. The queries that we saw so far were concerned only with data nodes. For queries with function nodes, the snapshot result is defined as for regular queries, except that the pattern embeddings map the function nodes in the query to function calls in the document.

We refer to tree patterns having `OR` and *function* nodes as ***extended*** queries. Examples are given in Figure 5.6. `OR`-nodes are labeled by $\oplus$ and function nodes have a gray background. In the following sections, we will use such extended queries to retrieve relevant function calls.

For compactness, rather than drawing tree patterns, we will sometimes use an XPath-like syntax for queries. In this syntactic representation, to distinguish data nodes from function nodes, we will add parentheses to the latter, as in the query `nyHotels/hotel/rating/getRating()`.

5.2 Finding relevant calls

Given a document d and a query q, our goal is to find a complete relevant rewriting, that is, a sequence of function invocations that makes d complete for q, and such that, at each step, the invoked function call is relevant w.r.t. the current state of the document. For simplicity, we will first ignore the function signatures and assume that functions can return arbitrary answers, i.e. that their output type is `any.` We will see in Section 5.4 how to use function signatures.

We start by explaining how to find, for a given state of the document, the function nodes currently relevant to q. We present two algorithms, starting with the simpler one, and compare their respective properties. Based on that, we then provide algorithms to find a complete relevant rewriting.

5.2.1 Linear path queries (LPQ)

The portion of a document that can be modified by the activation of a function is clearly determined, since the result is placed at the exact position of the function node. Therefore, a function node may be relevant for a query q only if it is on a path traversed by q. Based on this observation, we build a family of *extended queries* to retrieve relevant calls as follows: First, construct the *linear path queries* (LPQs) "embedded in q", i.e. the linear sub-queries of q. Then, replace their last node with a star-labeled function node, and mark this node as the output node of the query.

For instance, using the XPath-like syntax introduced in Section 5.1, the LPQs derived for the query of Figure 5.4 are:

```
/*()
/nyHotels/*()
/nyHotels/hotel/*()
/nyHotels/hotel/name/*()
/nyHotels/hotel/rating/*()
/nyHotels/hotel/nearby//*()
/nyHotels/hotel/nearby//restaurant/*()
/nyHotels/hotel/nearby//restaurant/name/*()
/nyHotels/hotel/nearby//restaurant/address/*()
/nyHotels/hotel/nearby//restaurant/rating/*()
```

The star-labeled function nodes (denoted `*()`) at the end of each query retrieve the function nodes residing at the end of a matching path in the document. It is fairly straightforward to see that, for any document d, *any* call in d that is relevant for q is necessarily returned by one of these linear path queries, when evaluated on d. However, what the LPQs actually compute is a *superset* of the relevant function calls. To see that, notice that on the sample document of Figure 5.1, the LPQs above select, among others, the calls to `getRating` and `getNearbyRestos` for the `"Pennsylvania"` hotel. These calls are in fact irrelevant, since the query is only interested in hotels named `"Best Western"`.

5.2.2 Node-focused queries (NFQ)

We need to use more sophisticated queries to get better accuracy. For that, we introduce *node-focused queries (NFQs)*. Instead of constructing one linear

NFQ(query q)
let $q_\oplus$ by a copy of q
for each node u in q
let u' be its counterpart in $q_\oplus$
replace u' in $q_\oplus$ by an OR between u' and f_u,
where f_u is a function node labeled $*$
end for
let $Q = \emptyset$
for each node v in $q_\oplus$
compute q_v as $q_\oplus - \{v$ and its subtree$\}$,
with f_v as output node
simplify q_v by removing redundant ORs,
add q_v to Q
end for
return Q

Figure 5.5: Building NFQs for a query q.

path query per node in the initial query, we derive a (potentially more precise) NFQ, that includes the *filtering conditions* from the original query. Notice that the latter can either be satisfied by data present in the document, or by data produced by the invocation of function calls. Thus, the NFQ for a given query node v will ask for all function calls found on the same path as v, such that all the filtering conditions could be satisfied either by some data, or by some (future) function call result.

The algorithm building all NFQs for a given query q is depicted in Figure 5.5. It starts by constructing a “modified query” $q_\oplus$, in which all nodes u are replaced by a choice between u and a star-labeled function node. Then, for each node v of q, the NFQ of v, denoted q_v, is obtained from $q_\oplus$ by erasing v and its subtree, and by marking the function call sibling of v as a return node.

Figure 5.6 shows three of the NFQs for the query of Figure 5.4: (a), (b), and (c) retrieve the function calls that might return relevant hotels, restaurants, and hotel ratings respectively.

Notice that, in these NFQs, `OR` nodes appearing in $q_\oplus$ on the path from the query root to the output node were removed, together with their `*()` branch.

This is because these branches cannot contribute bindings to the query result (and being under an OR, are not existentially required). Therefore, we can omit these nodes (in step 11 of the algorithm) without changing the semantics of NFQs.

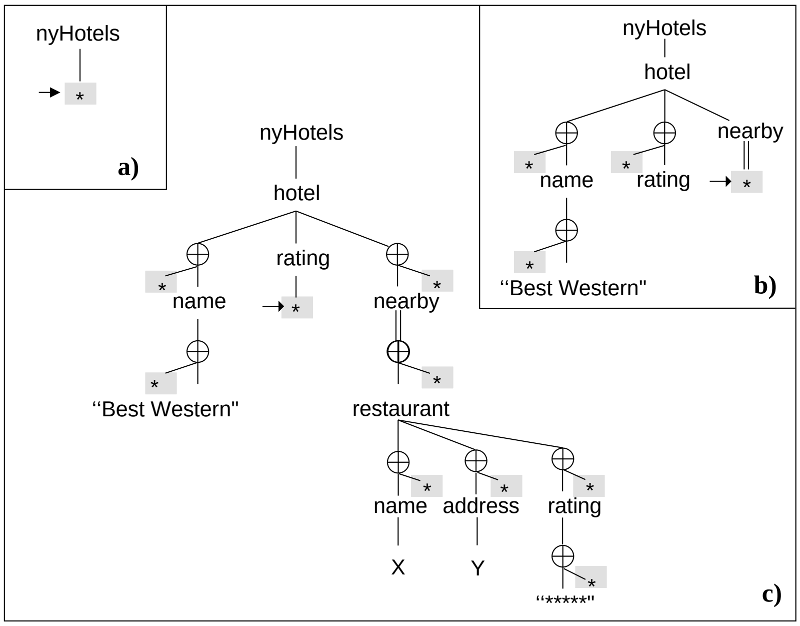

Figure 5.6: Three node-focused queries

The next property follows from the above discussion.

Proposition 5.2.1 *Assuming that functions can return data of arbitrary type, the function nodes in a document d that are relevant for a query q are precisely those retrieved by the NFQs of q, when evaluated on d.*

Proof: We first show that every function call returned by an NFQ is relevant. Suppose that for a given node v of a query q, the NFQ q_v returns call u on document d. By construction of the NFQ's, the linear path from the root of the query to v is matched in d.

Since functions are allowed to return data of arbitrary type, we are free to choose an answer for the invocation of u. Let the invocation of u return a tree that matches the sub-query of q rooted at v. For the other branches of

q_v, they are either matched by data present in d, or by function calls. Those that are matched by data are also matched in the original query. For the others, we similarly rewrite them into appropriate extensional data. This process yields a rewriting that starts with the materialization of u, is finite (only the functions initially present in the document are rewritten), and that leads to a document where q obviously has an embedding. Therefore u is relevant for q.

We now show that every relevant call is returned by a NFQ. Suppose u is a relevant call in document d for query q, and consider a rewriting that witnesses for the relevance of u, i.e. one that starts by invoking u and generates a node n transitively produced by u, that contributes to an embedding of q in d. Without loss of generality, since functions return arbitrary answers, we can assume that u returns an extensional answer and therefore that n is produced by u in one step. We can also take n to be the highest node in the tree that has this property. It follows that n must be the root of one of the trees returned by the invocation of u. Let v be the node of q that matches n in the above embedding. We argue that u is necessarily matched by the NFQ q_v in the original document d. This is obviously true for the linear part of q_v. For the other branches, since the rewriting leads to an embedding, the data matching q in the embedding is either present in the original document, or was produced by a function node that was invoked during the rewriting (as above, we can assume it was produced in one step). In both cases, since the NFQ has choices at every level between an arbitrary function node and a data node, this means that the data or the function call is matched by q_v in document d. As a consequence, q_v has an embedding in the original document, with u as an output node, therefore u is returned by q_v.$\square$

5.3 Sequencing relevant calls

We next explain how to derive a relevant rewriting, that is, a sequence of invocations in which *only relevant calls* are invoked. This is often an essential criterion, e.g., if we have to pay for calls. We will consider later some relaxations that simplify the analysis, at the cost of firing more calls than necessary.

5.3.1 The NFQA algorithm

NFQs induce a simple iterative algorithm for deriving a relevant rewriting: Given a document d_0, run the NFQs on d_0, pick *one* relevant function call f_0 among the returned calls, and invoke it, i.e. $d_0 \xrightarrow{f_0} d_1$. The next iteration will then run the NFQs *on the changed document* d_1, pick a relevant call f_1 among the results and invoke it, and so on recursively. We stop when no more calls are relevant, i.e. if for some step n, all NFQs return an empty answer on document d_n. We call this simple algorithm **NFQA** (for NFQ Algorithm). The next property follows from the fact that the algorithm invokes, at each step, a call that is relevant with respect to the current state of the document:

Proposition 5.3.1 *The NFQA algorithm computes a (possibly infinite) relevant rewriting. If it terminates, the obtained document is complete for the query q.*

Note that the potential non-termination is not due to the NFQA algorithm, but to the semantics of AXML documents/queries, that can indeed be infinite (see Section 5.1).

Clearly, through the execution of the NFQA algorithm, the NFQs stay the same, as they only depend on the query q. However, the *result* of the NFQs may change as the document evolves, thus they have to be re-evaluated at each iteration. The NFQs result may change in two ways:

1. The invocation of a call may bring new relevant calls, hence enrich the result of the NFQs.
2. Also, the invocation of a call may affect the document in such a way that formerly relevant calls are not relevant anymore, and this is what makes reevaluating the NFQs essential. For instance, if the `getRating` call of the `"Best Western"` hotel in Figure 5.1 is invoked and returns a low rating, the `getNearbyRestos` call, which was relevant before, becomes irrelevant.

To avoid the costly reevaluation of NFQs after each call, we analyze their mutual relationships. We present two optimizations, corresponding to items 1 and 2 above.

First, we characterize (in Section 5.3.2) the cases where the invocation of relevant calls, found by one NFQ, may return new relevant calls, enriching the result of another NFQ. This will enable us to apply a layering strategy, that splits the set of NFQs into smaller groups, on which NFQA is applied separately. Running NFQA on smaller groups *may yield much less* NFQ evaluations than doing so on the initial set.

Second, we detect some useful cases where a given call, once known to be relevant, is guaranteed to stay so (in Section 5.3.4). Assuming this property holds for two calls f_1 and f_2, we can invoke f_1 and f_2 *in parallel*, since we are sure that triggering one will not affect the relevance of the other.

Other optimizations are possible, since relevant rewritings are found by issuing NFQ *queries*. Their evaluation is delegated to a query processor, which can apply standard query optimization techniques [41], or eliminate redundant queries using containment checking as in [69]. The benefits of such techniques naturally apply in our setting. In particular, techniques for multi-query optimization [26] are essential to avoid performance penalties. In this section, we focus on techniques that are specific to our context.

5.3.2 Influence of NFQs

For a tree pattern query q and a node $v \in q$, we denote by q_v the NFQ of node v. For two nodes $v, v' \in q$, we say that q_v *may influence* $q_{v'}$ if the invocation of some call retrieved by q_v may bring new calls, detected by q'_v. Formally, we define influence as follows.

Definition 5.3.2 *For two nodes $v, v' \in q$, the NFQ q_v* may influence *$q_{v'}$ if there exist a document d, a function node $u \in q_v(d)$, and a rewriting $d \xrightarrow{u} d_1 \xrightarrow{*} d_n$, such that $q_{v'}(d_n)$ contains some function node transitively produced by u.*

For instance, the NFQ of Figure 5.6(a) may influence the ones of Figures 5.6(b) and (c), as the `getHotels` it retrieves may return, a `hotel` element with `getRating` or `getNearbyMuseum` calls in the right positions in its subtree.

We now explain how to detect whether an NFQ may influence another one. Recall that the result of a function call is placed in the document at the exact position of the call. Consequently, for an NFQ q_v to be able to influence another NFQ $q_{v'}$, the function calls selected by q_v should be *higher* in the tree than those selected by $q_{v'}$, so that the output of the former can actually be traversed by $q_{v'}$.

To state this formally, we use the auxiliary notion of *linear part of an NFQ* q_v, denoted q_v^{lin}, which is the linear path expression present in q, going from the root to v (not included). For example, q_v^{lin} for the NFQ of Figure 5.6(c) is `/nyHotels/hotel/rating`. Our algorithm for testing potential influence among NFQs is based on the following observation:

Proposition 5.3.3 *For a query q and two nodes $v, v' \in q$, the NFQ q_v may influence the NFQ $q_{v'}$ iff some word in the regular language of q_v^{lin} is a prefix of some word in $q_{v'}^{lin}$.*

Proof: If a word w in the regular language of q_v^{lin} is a prefix of a word w' in $q_{v'}^{lin}$, then a document witness for the possible influence of q_v can be constructed as follows. The document has one branch corresponding to w, ended with a function node u, and has other branches that satisfy extensionally (i.e. without any service call) the existential conditions of q_v and those of q_v'. Obviously, u is retrieved by q_v. Now consider the rewriting where u is invoked, and returns the remaining suffix in $q_{v'}^{lin}$, terminated by a function node u', together with branches that satisfy any existential conditions of $q_{v'}$ that appear on this suffix. The obtained document is such that u' is retrieved by q_v', and since u' was produced by u, it follows that q_v may influence $q_{v'}$. Observe that this construction is possible only because queries are monotone and contain no negation.

For the other direction, suppose that q_v may influence $q_{v'}$. Consider the corresponding witness document d, function node u and rewriting. Since u was retrieved by q_v, the path from the root of the document to u (not included) is necessarily a word in the language of q_v^{lin}. Since u is *replaced* by its result, any node transitively produced by u necessarily appears in a subtree below the same path, and so must be the function node u' that witnesses the *may influence* relationship. Therefore w is a prefix of the word w' that corresponds to the path that leads from the root to u', which is also a word in the language of $q_{v'}^{lin}$, since u' is selected by $q_{v'}$. This provides the construction of a word w in the language of q_v^{lin} that is a prefix of a word w' in the language of $q_{v'}^{lin}$, which concludes the proof.□

This proposition gives an immediate PTIME algorithm to check the influence relationship between NFQs: (1) build two regular automata, one accepting the words in q_v^{lin} and one accepting *the prefixes* of words in $q_{v'}^{lin}$, (2) test if their intersection is empty.

5.3.3 NFQ layers

We use the *may influence* relation among NFQs to split them into smaller groups, called *layers*.

Let $\leq$ denote the transitive closure of the *may influence* relation, and let $\equiv$ be the equivalence relation defined by $q_v \equiv q_{v'}$ if $q_v \leq q_{v'}$ and $q_{v'} \leq q_v$. Let NFQ *layers* be the equivalence classes of $\equiv$. $\leq$ induces a *partial order* between layers. To find a relevant rewriting, the interactions between NFQs can be analyzed based on their respective layers.

- By construction, NFQs inside one layer may influence each other, therefore, we need to run NFQA, as described in Section 5.3.1, inside each layer.
- For two *comparable* layers L_1,L_2, where $L_1 \leq L_2$ [2], we know that the processing of L_1 may augment the result of the queries in L_2, but the reverse is not true. Thus, we need to process L_1 first, and then we can process L_2.

This entails the following evaluation strategy. The partial order among layers is completed into some compatible total order. Layers are processed

[2]By abuse of notation, $\leq$ is also used for layers.

in increasing order. Within each layer, we apply NFQA. Note that when the processing of a given layer is over, we can simplify the remaining NFQs by removing the `OR/*()` branches corresponding to the layer we just finished. This is because the corresponding functions have already been invoked and cannot appear in the document anymore. This simplification makes the NFQs much less complex, hence faster to evaluate, without changing their results.

In our running example, the NFQ of Figure 5.6(a) may influence those of Figures 5.6(b) and 5.6(c), which are incomparable. Consequently, each layer consists of a single NFQ, with the one of Figure 5.6(a) being the first.

In general, layers may contain several NFQs. For instance, two NFQs $q_v, q_{v'}$ with linear paths $q_v^{lin} =$ `//a` and $q_{v'}^{lin} =$ `//b` would belong to the same layer, as paths that end with a `b` may have a prefix that end with an `a` and vice versa.

5.3.4 Parallelizing calls

The layering process described above decomposed the computation into smaller independent chunks. Let us now see how to further optimize this processing, by parallelizing the computation, *inside* one layer. More precisely, we want to know, given an NFQ q_v, under which conditions relevant function calls returned by q_v are guaranteed to stay relevant, independently of calling the other relevant function calls returned by q_v. In this case we can invoke all the returned calls in parallel and spare the re-evaluations of q_v that the naive NFQA algorithm needed after triggering each call. The following proposition gives a sufficient condition for that, which we call the *independence* condition. Whether this condition is also a necessary one is still an open question. As before, q_v^{lin} denotes the linear path expression going from the root of q to v, v not included.

Proposition 5.3.4 *Given an NFQ q_v, if for each NFQ $q_{v'}$ in the same layer as q_v, $q_v^{lin} \cap q_{v'}^{lin} = \emptyset$, then any call returned by q_v stays relevant if other calls returned by q_v are materialized.*

Proof: We prove that the above condition is sufficient for parallelizing calls by contradiction. Suppose that the condition holds, and that on a document d_0, $q_v(d_0)$ returns two function calls f_v and g_v, such that if $d_0 \xrightarrow{g_v} d_1$, then f_v is not relevant anymore, i.e. it is not in $q_v(d_1)$. This means that, for the embedding corresponding to f_v in d_0, g_v was matched by an $OR\{v', *()\}$ in an existential branch of q_v, and that g_v did not return data matched by the sub-pattern rooted at v' (or in other words, that the optimistic assumption about the output of g_v that made f_v relevant did not eventually hold). Consider the linear path corresponding to v', p_v^i in the original query. g_v is retrieved both by p_v and p_v^i, so $p_v \cap p_v^i \neq \emptyset$. p_v^i is the linear part of some NFQ $q_{v'}$. The two NFQ's q_v and $q_{v'}$ are necessarily in the same layer, since their linear parts have a non-empty intersection, hence the seeked contradiction.□

Note that this condition can be easily checked by constructing the regular automata of the two path expressions, and checking that their intersection is empty [53].

Going back to our running example, as each of the layers here contains a single NFQ, the NFQ is trivially independent and all the calls it retrieves can be invoked in parallel. The NFQA algorithm will then reevaluate the NFQ, to check if new calls were added by the previous invocations, and if so, will invoke the new calls in parallel, an so on. As another example, we saw a layer with two NFQs at the end of the previous subsection. As the intersection of their linear paths $q_v^{lin} =$ `//a` and $q_{v'}^{lin} =$ `//b` is empty, both NFQs are independent. For this layer the NFQA algorithm will, repeatedly, pick one of them, evaluate it and invoke all the retrieved functions, until no more functions are found.

We focused here on the maximum parallelism that can be achieved *without performing unnecessary calls*. Note that one may be able to reduce the time it takes to produce the answer by calling functions in parallel *just in case*, and thereby introduce more parallelism. This is an interesting research direction that requires the use of a cost model, and will not be considered here.

5.4 Using Types

So far, our analysis ignored function signatures. Let us now see how to use them to rule out more irrelevant calls.

The key idea guiding the construction of NFQs in the previous section was that, for each subtree of the query q, documents may either contain the subtree explicitly, or contain a function call that may return it. In practice, the possible shape of the data returned by a function is determined by its output type, and the output types of the functions that may appear in its output, recursively. To verify that the returned data may indeed be of the shape required by the query, the function signatures need to be analyzed.

Definition 5.4.1 *Given a schema τ, a function f defined in τ* satisfies *a query q if $q(d) \neq \emptyset$ for some* derived instance *d of f's output type, where the* derived instances *of a type are all the documents that instances of the type can be rewritten into.*

We will explain below how, given a function f and a query q, one can test if f *satisfies* q. For now, let us simply assume that such a satisfiability detection algorithm is given, and see how it can be used to refine the NFQs of Section 5.2. We will address this issue further.

Refined NFQs We use the following notation: For a query q and a node $v \in q$, sub_v^q denotes the query subtree rooted at v. We will also use, as previously, q_v to denote the NFQ query of v, and $q_v(d)$ to denote the result of q_v when evaluated on document d.

Recall that, for each node v, q_v looks for the functions that may contribute to the subtree pattern sub_v^q rooted at v. A natural filtering that one can apply to the functions retrieved by $q_v(d)$, is to discard those that do not satisfy sub_v^q (since these functions cannot contribute to the part of the query for which they have been selected, there is no point in invoking them). We can, for instance, discard all the `getNearbyMuseums` retrieved by the NFQ of Figure 5.6(b), since they return museum elements, and hence cannot satisfy the following subquery:

`//restaurant[name=X,address=Y,rating="*****"]`.

Note however that, while the above analysis can discard some irrelevant calls, it may still let irrelevant ones go through. Consider for instance the `getRating` call numbered 6 in Figure 5.1, which is retrieved by the NFQ in Figure 5.6(c). The function has a "correct" output type, since the ratings it returns can be matched by the original query. However, it is irrelevant, since the hotel's *nearby* element contains a function with inappropriate output type. To take this kind of typing information into consideration, the NFQs need to be *refined*, and must explicitly detail, for each query node v, which functions are "eligible", in the sense that they may actually produce data matched by the subquery sub_v^q rooted at v.

More precisely, given the list of all the function names retrieved by NFQs, we can test for each function f and each node v if f satisfies sub_v^q. Then, *refined NFQs* can be constructed as in the algorithm of Figure 5.5, except that rather than `OR`-ing each node $v \in q$ with a starred function node $*()$ matching any function, only the concrete names of the functions satisfying sub_v^q are listed. *Assuming that each function returns arbitrary instances of its output type, these refined NFQs retrieve precisely the set of functions relevant for* q *and* d.

For instance, the refined version of the NFQ of Figure 5.6(c) is depicted in Figure 5.7. Note that some of the function nodes have been removed (as there were no functions producing data of that type) and the $*()$ labels have been replaced by concrete function names.

A similar refinement can also be applied to the layering and parallelism processes of Section 5.3.

Refined relevant rewriting Recall that to optimize the NFQA algorithm, we ordered the NFQs so that those that may influence other NFQs will be evaluated first. Here again, the function type can be used to refine the layering. For a node v, let F_v be the set of functions satisfying sup_v^q. An NFQ q_v may influence another NFQ q'_v if some linear path of the first is a prefix of one of the second *and* the derived output instance of F_v may contain some F'_v functions. Similarly, when parallelizing the calls of a given

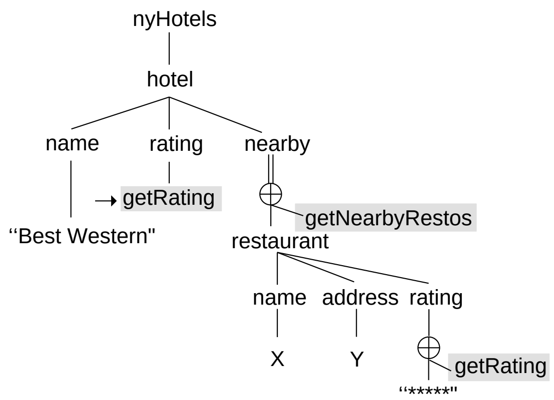

Figure 5.7: Refined NFQ

refined NFQ, it suffices to consider independence w.r.t refined NFQs that may retrieve the same functions.

Observe that during the rewriting process, when functions are invoked, their result bring new function calls into the document. The refined NFQs are enriched accordingly: the names of the new functions are added as possible alternatives to the query subtrees that they satisfy.

Testing satisfiability To conclude the discussion, we need to explain how, given a function f and a (sub)query q, one can test if f satisfies q.

To check satisfiability, one needs to test if $q(d) \neq \emptyset$ for some derived instance of f's output type. Observe that a similar problem has been studied for regular (non Active) XML documents: In [71], given a schema and a tree pattern query q, Milo and Suciu provide an algorithm for testing if $q(d) \neq \emptyset$ for an instance d of the schema. Their algorithm is based on (1) transforming the unordered tree pattern query into a set of ordered patterns by considering all possible orders/overlappings among the query subtrees and (2) checking the compatibility between the schema and the ordered patterns, using word automata techniques.

The difference here is that we are interested in testing satisfiability for the *derived* instances of the function output type, and not for its snapshot instances. This difference is significant: while the instances of a type are described by the regular expressions associated with the type name in the schema, the derived instances of a given type *can no longer be described by such a regular expression*. To see an example, consider a function f with output type $\tau_{out}(f) = a.f.b$. That is, f returns an a element followed by another call to f followed by a b element. The derived instances are of the form $a^n.f.a^n$, and thus belong to the class of context-free languages [53], and not regular languages.

Nevertheless, the derived instances of an AXML type can in general be described by an *extended* AXML schema, characterized as in Section 5.1, but associating a *context-free grammar* with each element name, rather than a regular expression. This context-free grammar is easily obtained from the schema, by treating each function call instance as a non-terminal, therefore using its type definition as a production rule. This extended schema can be constructed from the original one in PTIME. Then, the algorithm of [71] can be applied (with a slight extension) for the query and the function's derived output type, as defined by the extended schema.

The above technique yields an algorithm for testing function satisfiability, that runs in time exponential in the size of the schema and the query. One can further show that the problem is NP-hard even for very simple queries, i.e. queries without descendant edges, starred nodes nor joins (that correspond to repeated variable names). Therefore, it is unlikely that an algorithm with a lower time complexity exists. While this may seem negative, observe that the complexity is not with respect to the data size, but to the size of the schema and query, which are likely to be much smaller. Moreover, to further reduce the running time, an alternative is to use some lenient sufficient conditions, that may qualify more functions than actually needed, but can be tested more efficiently. We will see such a lenient algorithm, used in our implementation, in Section 5.5.1.

5.5 Faster relevance detection

In this section, we propose two complementary techniques to speed up the detection of relevant calls. The first one is based on a lenient variant of the algorithms presented in the previous sections, having a less refined but more efficient relevance detection component. The second one is based on a summary access structure that allows for fast pruning of irrelevant calls.

5.5.1 Lenient rewriting

The algorithm presented above guarantees that only relevant functions are invoked. To reduce the analysis time, one can trade *accuracy* for *efficiency*, running a somewhat more lenient (but faster) analysis, that invokes all relevant calls but possibly some more. In particular one can (a) relax the queries that find relevant function calls, and (b) relax the analysis of functions signatures.

Relaxed NFQ A substantial efficiency gain can be obtained by "approximating" each NFQ by a simpler query that can be evaluated faster. Two languages are natural candidates to approximate NFQs. The first one is *XPath*. XPath queries can express all variable-free tree patterns. Since NFQ's are themselves variable-free (they always return a single set of function calls), approximating an NFQ by an XPath expression essentially ignores the value-based joins. In return, the XPath approximation has lower time complexity and can be processed faster [47]. The second candidate language is *linear path queries*: the LPQs introduced in Section 5.2 can be seen as a relaxed version of NFQs, accounting only for their linear part.

Note that the NFQ layering and parallelism techniques presented in Section 5.3 can be used without modification for these relaxed queries.

Relaxed schema A crucial part of the analysis of function signatures is testing which sub-queries they can satisfy. To speed up this test, we use in our implementation a lenient description of the output types of functions, which ignores the cardinality of elements and their order. The derived output type of a function is then represented by a simple *graph schema*, in the spirit of [27], and checking satisfiability amounts to checking if the query can be embedded in this graph. This can be tested in time polynomial in the size of the schema.

A lenient relevance analysis can also be used as a pre-processing step, doing a first filtering of the irrelevant service calls before the "exact" analysis is applied. The benefits of such a preprocessing are illustrated next.

5.5.2 Function call guides

To further speed-up the processing, we use an access structure that provides information about the function calls that are present in the document. For efficiency, this information should be compact, and available in a way that is easily exploitable by the query processor.

In the spirit of dataguides [46], we use a tree structure that summarizes the paths that occur in a given document, containing a single occurrence of each path. The originality here is that we limit ourselves to paths leading to function calls. We call this structure a *function call guide* (F-guide for short). The F-guide holds, for each path, pointers to the corresponding function call nodes in the document. Figure 5.8 shows the F-guide for the document of Figure 5.1.

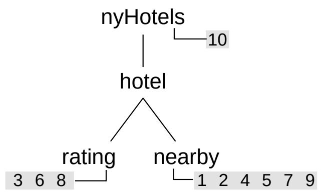

Figure 5.8: Function call guide

Since F-guides are trees, they can naturally be represented as XML documents, and therefore be serialized and queried just as the data they summarize. Moreover, this compact representation can easily be used to "approximate" the set of relevant function calls for a query. Indeed, it is easy to show that the linear path queries of Section 5.2 yield the same result on a document and on its F-guide. Therefore, rather than running the queries on the data, one can get better performance on its F-guide, which is typically much more compact.

Type-based filtering The above queries give an over estimate of the relevant calls, which can be refined using types. We can discard the functions whose output type does not satisfy the query subtree rooted at the end of the linear path by which they were retrieved, using essentially the same technique as in Section 5.4, or its approximation that was presented in the previous section.

NFQ filtering One can further filter the function call candidates using the NFQs. More precisely, for an NFQ q_v whose linear part is q_v^{lin}, the remaining query to evaluate checks for the conditions in q_v that do not appear in q_v^{lin}. This can be expressed as a relative XPath query, starting from the set of function calls returned by q_v^{lin}. For instance, for the NFQ of Figure 5.6(b), the remaining query is

`.[..[*()⊕rating][*()⊕name[*()⊕ "Best Western"]]].`

To conclude, note that the F-guide can be built in linear time in the size of the document, by a single (document-order) traversal. The F-guide must also be maintained as the document evolves. This maintenance must be performed if the document is updated, but also during query evaluation, when functions are invoked. The techniques to do that are similar to those of dataguide maintenance.

5.6 Pushing queries

In general, the result of a call may contain *more* data than is actually needed for evaluating the query. In such cases, to reduce data transfer and perhaps computation, it may be preferable to ask the function provider to send not the whole function result, but only the part useful for evaluating q. This amounts to *pushing* a query to the data source. For instance, in our running example, rather than invoking `getNearbyRestos` and getting all the restaurants nearby, we would like to also ship the subquery

`//restaurant[rating= "*****",name=X,address=Y]` (with `X` and `Y` marked as result nodes) with the call. Thus, instead of all vicinity restaurants, only the name and address of five-stars ones are returned.

Pushing queries is a fairly standard technique in mediator systems [84], and involves issues such as verifying that the remote source is capable of evaluating them [85]. We focus here on issues that are particular to our setting, namely how to determine which subquery needs to be pushed, and how to use the returned results.

Which subquery to push over a function call As previously explained, function calls are invoked when running the NFQA algorithm on a given layer of NFQs. We obtained f necessarily as the result of the NFQ q_v of some node v in q. The subquery to push over f is exactly the subtree rooted at v in the query, which we denoted sub_v^q in Section 5.4.

How to evaluate queries with pushed subqueries When functions are invoked, the call parameters and the query to be pushed are sent to the function provider. Rather than returning the full answer, the latter returns bindings to the query variables that are result nodes. For instance, when pushing `//restaurant[rating= "*****",name=X,address=Y]` over the call to `getNearbyRestos`, with `X` and `Y` marked as result nodes, the output consists of `X,Y` binding pairs, e.g.:

```
<tuple> <x>In Delis</x> <y>2nd Ave.</y> </tuple>
<tuple> <x>The Capital</x> <y>2nd Ave.</y> </tuple>
```

and not of `restaurant` elements. Subsequent queries that use this data (the NFQs to be evaluated in the future, and q) need to be modified to take these binding into consideration.

A first method is to actually *insert the result in the document* (replacing as before the function call) and modify the subsequent queries as follows. Let v be the query node for which f was discovered, and let $Z_1, \ldots, Z_k$ be the return nodes in sub_v^q. Then, in each subsequent query, replace sub_v^q by:

$sub_v^q \oplus \mathtt{tuple}[z_1=Z_1, \ldots z_k=Z_k]$

This approach is very simple, and follows naturally from the AXML model of query invocation. However, it has the drawback of leaving extraneous content in the document after evaluating q; `<tuple>` elements need to be eliminated afterwards. Alternatively, results from the service calls can be stored in an intermediary (temporary) storage, with the subsequent queries (the NFQs and the original query) modified to access them via *join*. This latter approach is indeed the one used in our implementation.

5.7 Implementation and experiments

All the ideas presented here have been implemented and tested in the AXML system. We used the algorithms described here to implement a module that decides which services need to be invoked for query evaluation, and pushes, when possible, the relevant subqueries to them. The system uses an XQuery-like query processor. The invoked queries are first *approximated* (in the sense of Section 5.5.1) by extracting their core tree patterns queries, which are then fed to the above module. The signatures of functions are given in their WSDL specification, as an (A)XML Schema, which we also approximate, as explained in Section 5.5.1 by a graph schema. This model is used for processing user queries as well as Web services defined by queries. Therefore, during the analysis of one query on one document, the same analysis may be applied, recursively, for the services invoked during the rewriting.

We ran several experiments to validate our techniques and briefly describe below a representative subset of them.

5.7.1 Experimental setting

Hardware and software environment As a Web service communication infrastructure, the system uses the SOAP implementation of the Apache Axis platform. All the implementation (including the XML query processor) is done in Java. We performed the measures on a Compaq Evo N410c, with 256 MBytes of RAM and a PIII 1.2 GHz CPU, running Mandrake Linux 9.2.

Documents and functions used The documents we use have a structure very similar to the example in Figure 5.1. They all consist of `hotel` elements, which may include calls to the functions: `getRating`, `getNearbyRestos`, and `getNearbyMuseums`; we also use an extra `getNearbyHotels` function whose result is a set of hotels. The latter may contain similar calls, thus requiring a recursive relevance analysis.

We used the ToXgene [18] XML generator to produce documents. We fixed the document size to be about 1.6MB, and varied the number of function calls in the document by instructing ToXgene to set the content of, e.g., `rating` elements f % of the time as a call to `getRating`, and (100-f) % of the time as materialized data. The same fraction f controls the probability of a `nearby` element to contain a call to `getNearbyHotel/Restos/Museums`, respectively, instead of plain data without function calls. The following table summarizes the parameters of the generated documents:

Value of f (%)	2	20	27	35
Number of function calls	466	757	970	1346

Queries We report experiments performed with two queries:

```
/hotels//hotel                      (LIN)
/hotels/hotel[rating=‘‘***’’]       (TRE)
```

We chose these queries to illustrate a variety of scenarios. The linear query `LIN` is quite simple, but it searches at any depth in the document. `TRE` is a tree query, with branch tests at all levels, following only precise paths.

5.7.2 Experimental results

Some results of our experiments are shown in Figures 5.9 to 5.12. In all of them, on the x axis we vary the number of service calls. We consider four different filtering scenarios, using: only an F-guide; the F-guide and the type analysis; the F-guide and NFQ filtering; and finally, all three filtering techniques. We used two approximations: NFQs expressed in XPath, thus with no value joins; and relaxed schemas as in Section 5.5. For this particular experiment, these approximations happen to be precise.

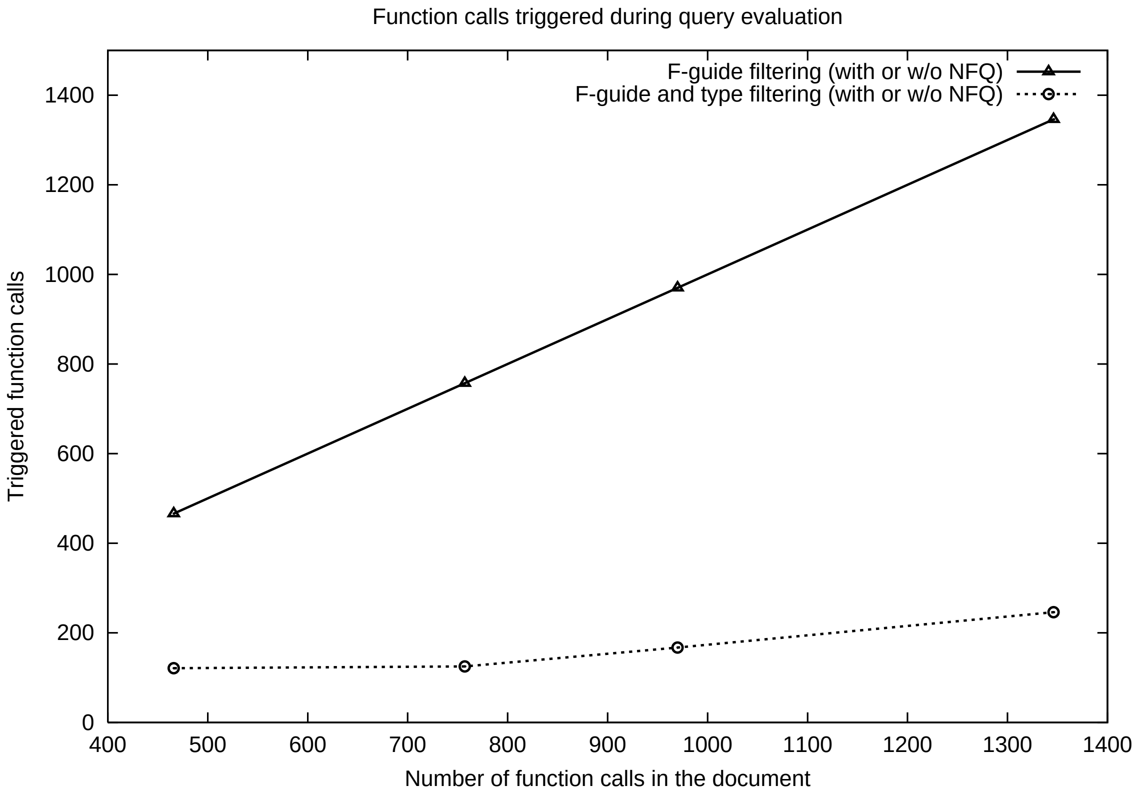

Figure 5.9: The effect of type filtering (query LIN)

Number of triggered service calls In Figure 5.9 (resp. Figure 5.10) , we counted the calls triggered by the evaluations of `LIN` (resp. `TRE`) In the case of `LIN`, F-guide filtering does not eliminate any call, since the query traverses the whole document: thus, all function calls are triggered. Type-driven filtering, on the other hand, achieves a significant reduction, since it avoids the useless invocations of `getNearbyMuseums`, `getNearbyRestos` and `getRating`. The NFQ technique does not alter the impact of either F-guide, or type filtering, since `LIN` has no branch conditions.

For `TRE`, F-guide filtering leads to ignoring the calls found in regions of the document not traversed by the query; still, all calls under `hotel` ele-

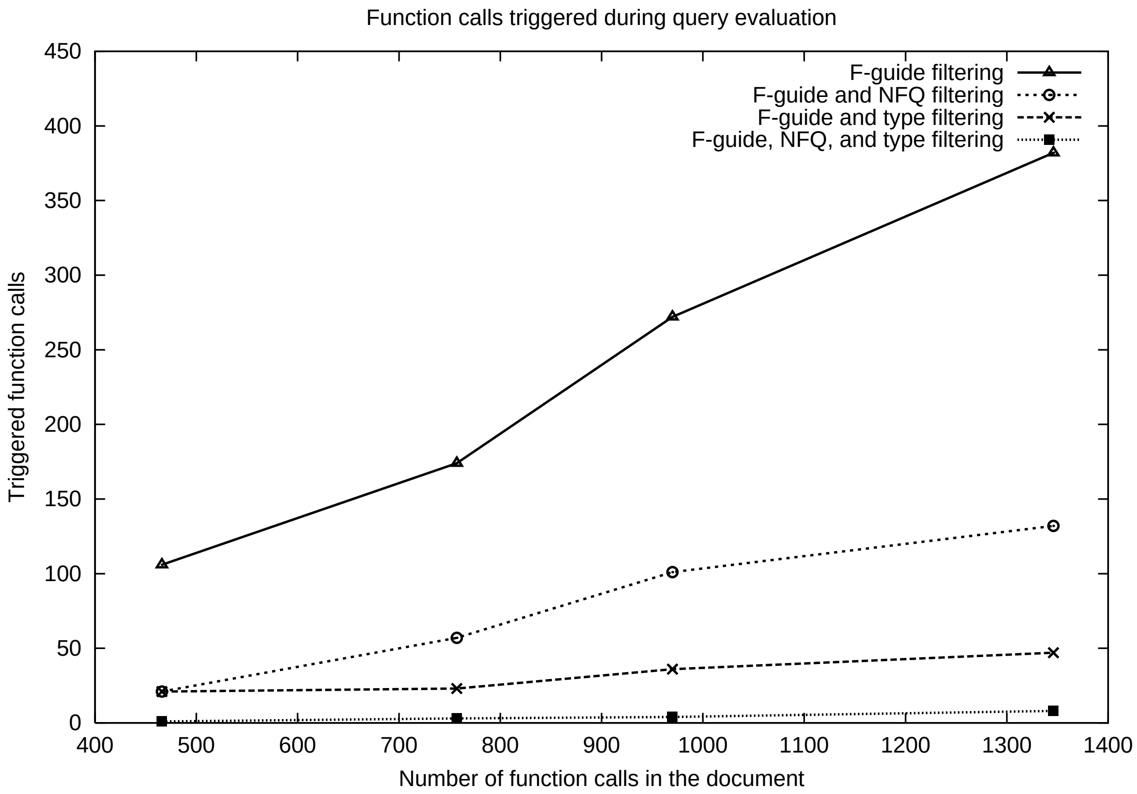

Figure 5.10: The effect of call filtering (query TRE)

ments, whether they bring ratings, hotels, museums, or restaurants are triggered. NFQ exploits the predicates on rating to avoid triggering such calls for hotels having the wrong rating. Type filtering eliminates irrelevant calls to `getNearbyMuseums` and `getNearbyRestos`. The cumulated effect of the three techniques leads to the best performance, with for instance only 8 out of 1346 function calls fired.

Analysis time In Figure 5.11, we measured the total time spent doing relevance analysis for `TRE`, that is, building the F-guide and analyzing it, testing function signatures, and evaluating NFQs. F-guide construction and analysis is very fast (the bottom curve); F-guide and type filtering is slightly more expensive. Running NFQs takes some time, therefore the curve for all three filtering techniques is higher. Note however that the three bottom curves do not vary much with the number of service calls. The slight W-shape

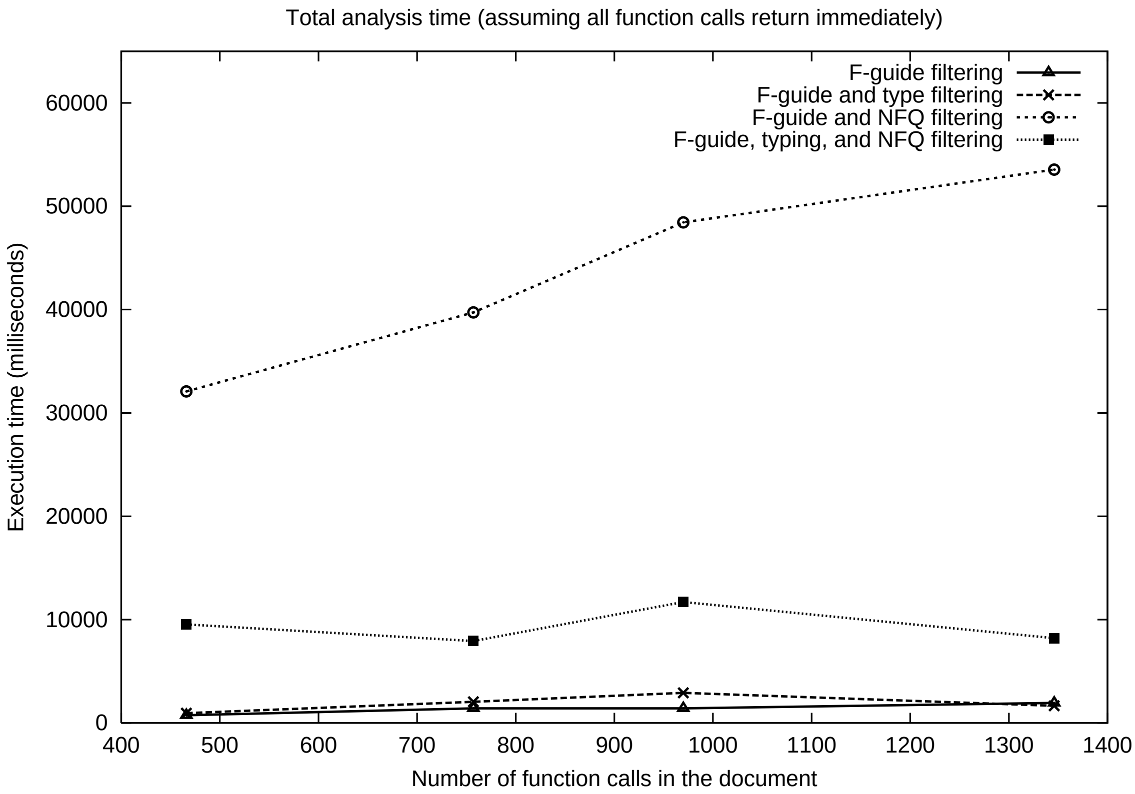

Figure 5.11: The impact of call filtering on analysis time

of the three-filter curve is due to a data variation in the XML documents that we were not able to control – the co-occurrence of three-star hotels at three successive levels was higher in some documents than in others. Using F-guide and NFQ filtering, but without checking the types (top curve) is a very bad choice: it leads to evaluating a large amount of NFQs, for many irrelevant calls to `getNearbyRestos`, `getNearbyMuseums` etc. This curve grows linearly with the number of services in the document.

Total evaluation time The graph of Figure 5.12 measures the total evaluation time, namely analysis *plus* service invocations, to capture the tradeoff between the time spend on analysis vs. the time saved by avoiding useless invocations. We set the function call duration to 500ms. This is a very minimal value, since the times we measured for calling a no-effect service call (constructing an empty-content outgoing SOAP message, sending it to the

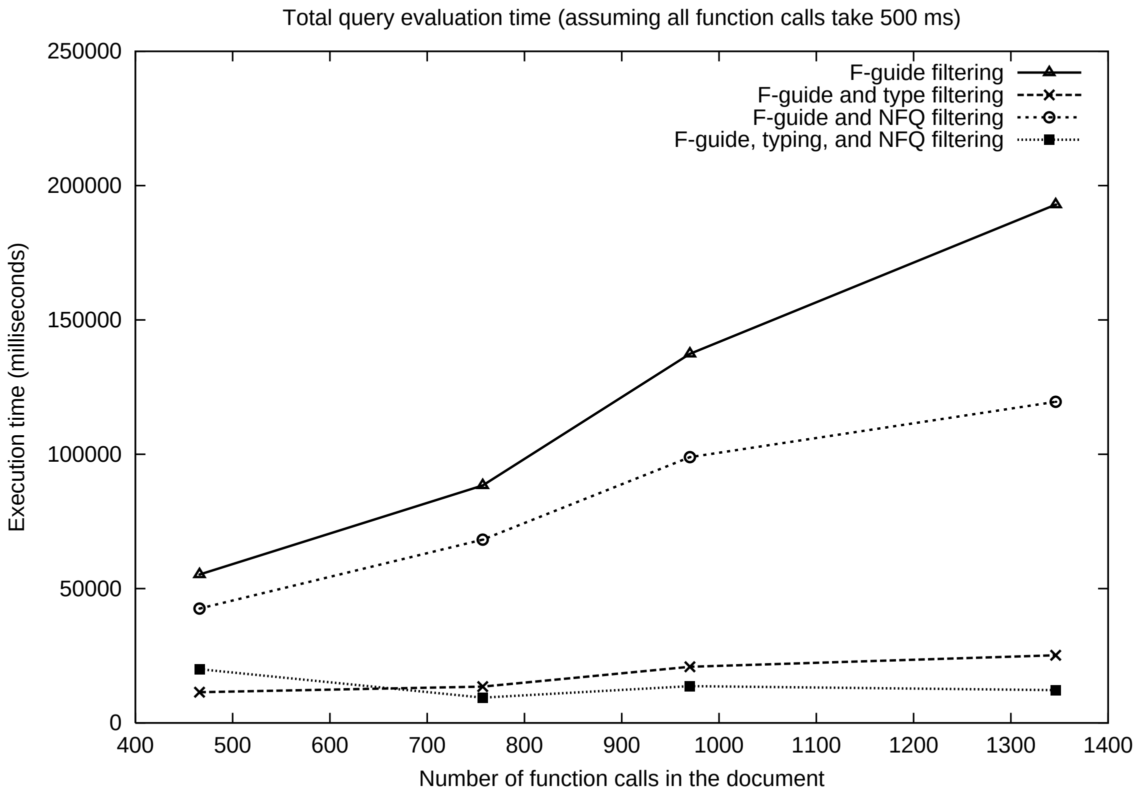

Figure 5.12: The impact of call filtering on query evaluation time

same host, sending an empty answer back, unpacking the message) varied in average between 500ms and 2500ms. In practice, a service call is likely to last much longer. In this graph, F-guide analysis alone becomes extremely expensive, due to the large number of function calls it triggers. F-guide and NFQ filtering is somehow better, since NFQs are effective in pruning function calls for `TRE`; still, in the absence of type checking, it is very expensive. F-guide and type filtering drastically reduce the total query evaluation time; adding NFQ to the mix brings a further slight improvement. The lowest two curves cross, when the extra time spent by the three-techniques analysis pays off the time saved in function calls.

Similar results are obtained for query `LIN`.

To summarize, NFQ filtering is useful for queries having branch conditions, and does not affect the others. Simple F-guide analysis, without considering the functions' return types, is a rather bad choice, since it results

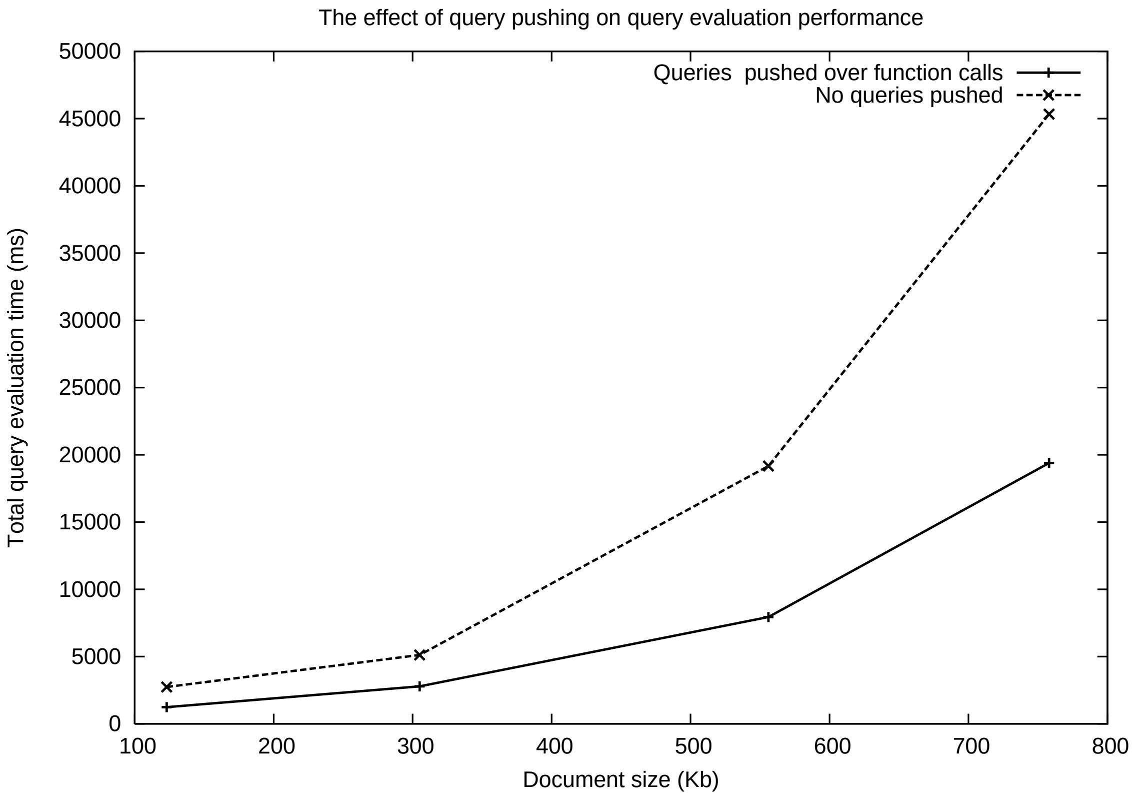

Figure 5.13: The effect of pushing queries

in many irrelevant calls. Finally, and most importantly, F-guide, type, and NFQ filtering, together, are extremely cost-effective: the analysis pays off very significantly, both in terms of function calls and total evaluation time. Combining all filtering techniques, we observed more than a 10-fold reduction in the total evaluation time of `TRE`; in a more realistic setting, where Web service calls often last more than half a second, the improvement is likely to be even more important.

The effect of pushing queries We also measured the effect of pushing subqueries over function calls. Figure 5.13 depicts the average total query evaluation time for queries in the style of `TRE`, with and without pushing. As expected, data transfers are greatly reduced (up to three times on our fast LAN). The curves are quadratic due to the fact that the *getNearbyHotels* service is implemented by a query over the hotels document. The more hotels the document contains, the more nearby hotels there are, and the query will run on a document of size quadratic in the initial number of hotels.

5.8 Conclusion and related work

In this chapter, we presented an algorithm to efficiently find and invoke the service calls relevant to a particular query, taking into account the output types of services, and pushing queries to them when possible. We also introduced an auxiliary access structure, the F-guide, to speed up the processing and presented results of experimental studies.

The evaluation of queries over AXML documents with lazy service calls is also the topic of [10]. Their focus is on minimizing data materialization at a global level (using a generalization of Query-Subquery [99]) for systems with many interrelated documents in a peer-to-peer setting. Also set in the context of AXML, a previous work [7] considered the problem of distributing and replicating AXML data and services in a peer-to-peer setting. In contrast to both, we addressed here the specific problem of optimizing the evaluation of queries over *one* local AXML document. So, our work is clearly complementary to both, and indispensable for them to obtain good performances.

AXML documents may be seen as a means of integrating data sources, that combines mediation [84, 67] and warehousing [49]. There are indeed analogies between finding relevant calls and pushing queries to them, as proposed here, and query rewriting using views [50]. A novelty in our approach is that mappings between data sources are captured by service calls embedded in the data, with new relationships discovered at run-time, in the answers of service calls. This approach, based on ad-hoc dynamic discovery of mappings, is in the spirit of peer-to-peer data integration systems [51]. However, in systems like [51] the integration happens at the schema level, while in AXML it happens at the data level. This strongly suggests combining the two approaches.

Our F-guides are inspired by dataguides [46]. Although they serve different purposes, they are both based on paths in the original documents. This is also in the spirit of adaptive path indices [33]. Invoking calls only when needed is a technique classically used in programming languages, under a variety of names, e.g., lazy reduction for lambda calculus [19]. The originality of our work comes from the particular setting, XML with embedded service calls.

We already mentioned various techniques that we believe are candidates to meet ours towards solving the general problem of query processing for AXML. It would also be interesting to consider techniques for minimizing communication, e.g., semi-join-based techniques. One should also consider an important classical aspect in mediator systems, namely source capabilities. This is also an issue in our context, for pushing queries to sources, and descriptions of the sets of queries accepted by services, such as proposed in [85] could be used. In the spirit of peer-to-peer computing, one should also consider the issue of publication and discovery of data sources, see [97]. Issues such as quality of service, unreachability of some peers, substituting one data source for another should also be considered.

Chapter 6

AXML peers

In the previous chapters of this thesis, we presented the AXML language, based on AXML documents and AXML services, then studied two important issues: the exchange of AXML documents, and the lazy evaluation of queries on them. To effectively support this language, we next adopt a distributed architecture based on the *peer-to-peer* paradigm, where each participant may act both as a client and as a server. The systems that participate in this architecture are called *AXML peers*. The topic of this chapter is their features and implementation.

Since the AXML language is based on the use of Web services, our architecture relies on invocations of Web services, for all communications between participating peers. AXML peers have essentially three facets:

- **Repository:** The first task of an AXML peer is to manage persistent information, very much like a database system. Since our goal is to support the AXML model, data will naturally consist of *persistent* AXML documents.

- **Client:** In order to take advantage of the intensional data present in its repository, an AXML peer may invoke the corresponding Web services. By doing so, it acts as a client of other peers. We extend the AXML language with features to control the activation of service calls.

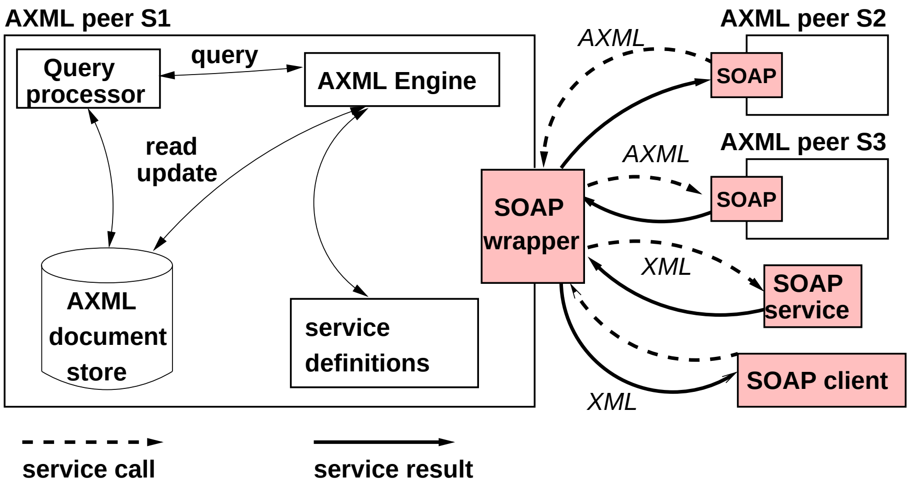

Figure 6.1: The AXML peer-to-peer architecture

- **Server:** An AXML peer may also provide Web services for other peers. These will typically be defined using queries or updates over the documents of its repository, and will indeed be AXML services.

The AXML architecture is illustrated in Figure 6.1, which shows the main internal components of an AXML peer. AXML peers exchange AXML data with each other through their Web services using the SOAP protocol. Existing Web services (such as Google) and clients (such as the Mozilla browser) naturally fit in this architecture, since all exchanges follow standards for Web services. However, the exchange of AXML documents will only occur between systems that understand the language.

This chapter is organized as follows. The two first sections are devoted to the functionalities of AXML peers, first as clients, and then as servers. Section 6.3 will overview the implementation of an AXML peer, while Section 6.4 will illustrate the use of AXML peers through application scenarios.

6.1 AXML peer as a client

The main issue when viewing an AXML peer as a client, is how to manage *persistent* AXML documents, while taking advantage of the service calls they contain to enrich them. For instance, on the newspaper document of Chapter 3, we may want to enforce the following behavior:

> *Temperature information should be refreshed daily. New exhibits should be fetched every week and archived for 6 months.*

Note that in this setting, service calls should be *kept inside* AXML documents, for future reuse. Therefore, materialization will not replace service calls by their results anymore, but will append the results of each call next to it. While this is a slightly different semantics than the one used in previous chapters, it is necessary in the context of an AXML peer managing persistent data. We should mention that this change of model does not affect the algorithms of lazy evaluation presented in Chapter 5 (calls are marked as executed instead of being removed). For the typing algorithms of Chapter 4, calls are indeed replaced by their results (since the main goal is to hide the intensional definition of data), but recall that these algorithms are mainly used in the context of exchange of information, and therefore only affect the exchanged messages, and not the persistent data.

The main novelty in managing persistent AXML documents is the evaluation of calls, which entails a number of issues: (i) when to activate a call, (ii) where to find its arguments, and (iii) what to do with its result. These issues are considered next in turn.

When to activate a call?

Consider a service call in a document of a peer c (for client) that calls a service on a peer s (for server). One can first distinguish between calls that are activated from the client side (*pull* mode), which are regular function calls, and those activated from the server side (*push* mode). For the pull mode, we will further distinguish between what is called explicitly (*explicit pull*), e.g. because of a user request, and what is called implicitly (*implicit pull*), because some other operation required this call to be activated. The particular form of activation control that is used is specified using a special attribute of the *call* element: the `mode`, such as in:

```
<call svc="..." mode="daily">...</call>
```

Let us now consider these alternatives in more detail.

Explicit pull The most standard type of explicit pull is based on time frequency, e.g., activate this particular call daily, weekly, etc. For instance, we may want to call a Yahoo Web service daily to get the temperature information. More generally, one can request the call to be activated after some particular event occurred in the system, e.g., when some other call has completed.

This aspect of the activation problem is closely related to the field of active databases, where triggers are used to update the database whenever some particular events occur, and/or some conditions are satisfied.

Implicit pull This means that we want to call a service only when the data it returns is needed by a computation the system needs to perform, typically. Implicit pull is also called the *lazy* mode, because it is in the spirit of lazy evaluation of calls in functional programming. In case the computation to perform is a query, the algorithms presented in Chapter 5 are used to determine and activate the calls that are relevant to answer that query. In the system, these algorithms apply only to a subset of the service calls embedded in the documents of the repository: those that are specified as lazy, again using the `mode` attribute of the *call* element.

This aspect of the activation problem is closely related to the field of deductive databases. This is because of the recursive nature of the context. We need to generate only the "relevant data", which is in the spirit of the goals achieved by query optimization techniques for deductive databases such as Magic Sets [98] or Query-SubQuery [99].

Push In push mode, the server decides to push information to the client, generally because the latter subscribed to some continuous service it provides. It is then the choice of the server to decide when to send information to the client. A stream of data from the server will then be flowing to the (subscription) service call node of the client document. As we will see, additional parameters can be specified in the *call* element, to control how the continuous service should operate. These will be considered in Section 6.2.

In our example, the local newspaper may subscribe to exhibit announcements provided by the TimeOut local guide, and be notified each time a new exhibit opens in the city. Push services can also be useful for managing change control, and in particular for synchronizing replicas of the same data. This aspect of the problem is closely related to stream management and subscription queries.

How long will the returned data remain valid?

Another important issue when managing persistent AXML documents is to control how long some data remains valid (i.e. persists), once it has been obtained. This is also guided by an attribute of the service call, called `valid`, which may take a number of values:

1 day, 1 week, 1 month, etc. This may be viewed as some controlled form of caching.

Validity is zero This means that the data remains valid only for the time needed to serve the request that asked for it. This is in the spirit of mediators, that do not store any information but simply obtain it when needed. It is generally used together with a lazy mode, where calls are activated when the data they retrieve is needed to answer a query.

Unbounded The information will remain present forever, unless explicitly deleted. This can be used in particular for archiving purposes, where data is never deleted. However, unbounded validity may also be of interest in other settings. For instance, when used together with a replace mode (the information is erased when a new version is obtained, see further), it allows to keep a single version of the result, that is, the last one. Also, it may serve as the basis for some form of caching, if used together with other services that explicitly delete data to reclaim space when needed.

```
<?xml version="1.0" ?>
<newspaper>
  <title>Le Monde</title>
  <city>Paris</city>
  <date>06/10/2003</date>
  <weather>
     <call svc="Yahoo.GetTemp" mode="lazy" valid="1 day">
       <city>Paris</city>
     </call>
  </weather>
  <exhibits>
     <call svc="TimeOut.GetEvents" mode="every Monday morning"
                                   valid="6 months">
       exhibits
     </call>
  </exhibits>
</newspaper>
```

Figure 6.2: A persistent AXML document

Note that various portions of an AXML document may follow different policies for validity, based on the nature of the information. For instance, in comparative shopping applications, one often needs to combine a mediator style for some products that change price very rapidly (e.g., plane tickets) and a warehousing style for other products (e.g., housing).

To illustrate the use of attributes controlling the execution of service calls, the newspaper document may be written as in Figure 6.2.

In this example, the service call that gets the temperature is called when the data is needed, only if the data is older than 1 day, whereas the one that gets the list of events is called weekly, and data stays valid for 6 months.

What to do with the result?

When service calls are called repeatedly in persistent AXML documents, one needs to define the interaction between the answers returned by an invocation and those obtained from previous invocations. We first consider a local insertion, and then a more global fusion. As before, attributes of the *call* element are used to specify what to do with the result.

Local merge In the example, we assumed that the result of a service call was inserted next to it (as a sibling of the *call* element). More precisely, a call returns an AXML forest and when the call is reactivated, we need to know what to do with its result and with the existing forest (the former siblings of the service call). To do that, we use *merge functions*. Some are provided by the system but more may be defined, locally or as Web services. The merge functions provided by the system are:

- **append:** just append the new forest next to the previous one. This is the default mode;
- **replace:** replace the old forest by the new one;
- **fusion:** Based on IDs or key specifications (such as those of XML Schema), identify the "same" elements among the old AXML forest and the new one and merge them. Clearly, there are various ways of merging two elements. The system provides the choice between an append or replace strategy, and also supports nested fusion rules. Other kinds of fusion could be used by introducing customized merging operators.

Global merge The same mechanisms as for local merge are used, except that the new forest is merged with the whole document and not only with the siblings of the *call* element. It is important to observe that, with global fusion, it is much harder to control which portions of the document are affected by a service call invocation. Therefore, the system currently forbids the use of these calls together with the lazy mode.

To conclude this section, we should note that service calls may lead to errors. This is because some merges may fail, e.g. due to key specifications that do not match the data, or to a duplicate attribute inside an element that makes the document not well-formed anymore. In such cases, the call that is responsible for the error is simply rejected, and its results are ignored.

Where to get the call parameters?

Recall that the arguments of a service call are specified as children of the *call* element. In the simplest case, an argument is an XML tree. More generally, it can be any AXML element, therefore it may itself contain some service calls. Indeed a particular case of services turns out to be very useful in this context, namely those defined as XPath queries. For instance, instead of stating explicitly that we want the temperature of Paris, we may request the temperature of `"../city"`.

Note that this is actually a regular call to a local Web service, which is defined using an XPath query. Relative path expressions (evaluated starting from the *call* element) translate to queries that refer to the unique ID of the corresponding service call in the document. The fact that we allow to specify the XPath expression directly is just useful syntactic sugar.

Such XPath parameters are evaluated in lazy mode, whenever the service call they correspond to needs to be invoked. The algorithms of Chapter 5 apply in this setting, to find and invoke any calls (marked as lazy) that may be relevant to answer the XPath query that retrieves the value of the parameter. If calls retrieved by this analysis also have XPath parameters then, recursively, another lazy evaluation of these parameters will be performed.

Note also that such an XPath parameter cannot be passed intensionally to the invoked Web service, because it only makes sense in the context of this particular node. Thus, such context dependent calls have to be resolved before calling the service. In general, when the parameters of a call contain some unresolved service calls, it is unclear whether we should evaluate them or not before sending. Some have to, e.g., path expressions that make sense only locally. For others, a particular choice may influence the semantics of the application. In the system, we use the algorithms presented in Chapter 4 to control this materialization, based on the input type specified by the Web service.

6.2 AXML peer as a server

The previous section showed how an AXML peer can activate calls to Web services embedded in the persistent AXML documents it manages, and various flexible ways to control this behavior. But a peer may also support and publish Web services itself, as illustrated in this section. Recall from Chapter 3 that AXML services are Web services that exchange AXML documents. In short, AXML services are mainly defined by parameterized queries or updates over the peer's AXML documents. The system also supports other languages for the definition of services, and allows for push services.

Pull: query and update services

The basic way to define a service in an AXML peer is to write a parameterized query over its repository of AXML documents. Here is a sample service definition [1]:

```
let service GetExhibitsByLocation($loc) be
  for $a in document("newspaper.xml")/newspaper/exhibits,
        $b in $a//exhibit
        where $b@name=$loc
   return <exhibits> $b </exhibits>
```

Figure 6.3: An AXML Service definition given as a query

This query uses the `newspaper.xml` document from the repository, that may be similar to the one of Figure 6.2. When a call to this service is received by the peer, the `$loc` variable is bound to the transmitted parameter value (the location), then the query is evaluated, and its result is returned as an answer. It should be noted that any kind of AXML fragment can be passed as a parameter to the service (e.g., a service call, a forest of AXML trees, or an atomic value).

[1] The queries in the AXML system are written in the X-OQL language [12], but we use here the syntax of XQuery, which is more commonly used and is currently becoming the standard language for querying XML documents.

Again, the lazy evaluation technique of Chapter 5 will be leveraged, in order to find and invoke the service calls that are relevant to the query that defines the service, both in the repository documents and in the parameters passed by the invocation.

Also, note that the query answer may naturally contain service calls, since the query is evaluated on the AXML parameters and the AXML documents of the repository. The typing algorithms presented in Chapter 4 will be used to make the result match the output type of the service. This output type is generally specified in the WSDL description of the service, that is, by the service provider. In our implementation, we also allow the client to dynamically specify a desired output type, as an extra parameter of the service invocation.

We also support update services, that have side effects on the repository documents. These are also defined declaratively, as an extension to X-OQL, that is very much in the line of the update extension to XQuery proposed in [95].

Providing such services assumes of course that the peer has the appropriate query/update engine. More limited AXML peers, such as those running on PDAs, provide less complex services, defined using XPath queries for instance. Such a lightweight AXML peer was indeed implemented as part of the AXML project [36].

Pull: Other kinds of services

Other languages are sometimes more appropriate to define services than query/update languages. We integrated two important ones into the AXML system:

- **Workflow services:** AXML peers can provide access to services defined using the Business Process Execution Language for Web services (BPEL4WS) [25], one of the main proposals to define a workflow language for Web services. To execute such services, AXML relies on the implementation provided by IBM, and enriches them with the ability to return AXML answers whose type is controlled using the algorithms of Chapter 4.

- **XSLT services:** Services can also be specified using the XSLT transformation language [113]. Such services take a single parameter, which is an XML document, and return the result of transforming the input document using the stylesheet specified in the definition of the service. While XSLT uses XPath as a selection language, to extract information from the document to transform, in the current implementation, such services do not take advantage of the intensional information that may be present in the input AXML document, since we have not integrated the technique of Chapter 5 within the XSLT processor that we use for the transformation.

Push: continuous services

A server may work in pull mode, as described above, but may also work in push mode: The client subscribes to a particular push service, by issuing a service call, then the server asynchronously sends a stream of messages as an answer.

These services may also be specified declaratively, e.g. using parameterized queries. Additionally, a number of parameters specify how such a service operates:

- a frequency of the messages (e.g., daily);
- some limitations, such as the duration of the subscription or some size limit on the data that is transmitted in each message;
- a choice of representation for the changes: send consecutive versions, send a delta or an edit script for the changes, see e.g., [35], send a notification and publish the changes.

Typically, the AXML peer supports a limited subset of these choices among which the client chooses, by specifying the corresponding parameters in the subscription call.

6.3 Implementation

We now overview our implementation of the AXML peer. Since many people in the Gemo team contributed to the implementation of this system, we only give here a description of what we mainly contributed to and implemented, namely the general architecture of the system. Before that, we start with an overview of the involved technologies.

Technical environment

We start by presenting the technical environment we chose for implementing the AXML peer. Figure 6.4 presents the stack of technologies the system is built upon. Its main components are as follows.

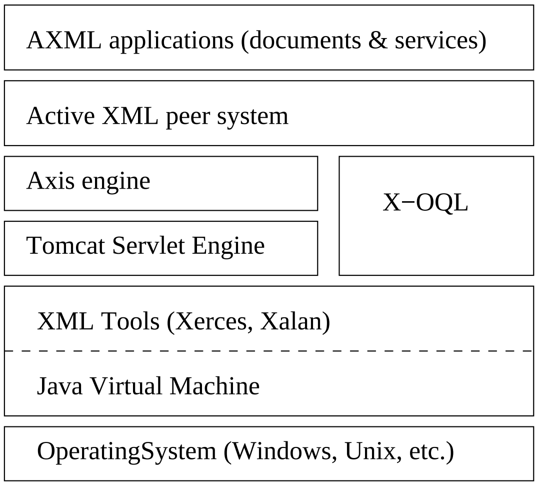

Figure 6.4: The AXML peer technology stack

The Java Virtual Machine The AXML peer is implemented in Java [59], one of the leading and most widely used programming languages today. Java programs are portable, since they run on top of a Virtual Machine, which abstracts the details of the underlying operating system. Java Virtual machines exist for most modern operating systems (Microsoft Windows, GNU Linux, SUN Solaris, Apple Mac OS X, etc.).

XML tools Since AXML is based on XML, tools are needed to process XML data. The AXML peer relies on the Apache Xerces validating XML parser to parse documents, and manipulate them through their representation in the Document Object Model (DOM) [39]. It also relies on the Apache Xalan processor to run XPath queries and XSLT transformations on them. In Java, XML processing is normalized by the Java API for XML Processing (JAXP). Since the tools we use are compliant with this specification, they can be seamlessly replaced by other packages that provide the same functionality and also follow the JAXP conventions.

The X-OQL query processor As mentioned already, the AXML peer relies on the X-OQL processor [12] to execute complex queries on XML documents. X-OQL operates on a DOM representation of XML documents. In the current system, this DOM is generated by the Xerces parser, when documents are read from the filesystem, and is stored in main memory. This is clearly an issue when documents become very large. We are considering the use of a disk-based virtual DOM, exposed by a native XML repository, and X-OQL is being tested in this setting at the University of Marburg [114]. We are also considering the implementation of an AXML peer on the Xyleme Zone Server [115] native XML repository and query engine.

The Tomcat servlet engine In order to provide services accessible over the Web, the AXML Peer needs to act as a Web server. This role is fulfilled by the Tomcat engine [96], a servlet container. Servlets are Java programs invocable over the Web, using the HTTP protocol. A Tomcat engine can host multiple independent Web applications, called Webapps. So technically, the AXML peer is a Tomcat Webapp. Tomcat also supports the Java Server pages (JSP) [62] templating language, to generate dynamic Web pages. In the AXML peer, JSP pages are used together with XSLT transformations to generate its Web-based management interface, and to easily develop a user interface for AXML applications.

As for the XML processing tools, the role fulfilled by Tomcat is normalized by specifications, namely the Java Servlet and JavaServer Pages specifications [2]. Therefore, Tomcat can be seamlessly replaced by alternative, more robust application servers, such as BEA WebLogic, IBM Websphere, or Oracle iAS, that implement the same specifications.

The Axis SOAP toolkit Axis [15] is a Java toolkit designed to ease both the implementation of Web services and the access to existing services from Java programs. In other words, Axis enables Web services functionality both on the server-side and the client-side. The AXML peer relies on the Axis toolkit whenever service calls need to be issued or answered. By doing this, the AXML peer leverages Axis to ensure that communications strictly follow Web services standards, and are therefore interoperable with other service providers and consumers.

The AXML peer also builds on the flexible architecture of Axis for message processing, which is based on *chains of handlers*. A chain is simply a sequence of handler invocations, where each handler is a program that performs some part of the required processing. Handlers share a common message context, that is used to pass information between them. Chains of handlers are easily configurable on a per service basis, without the need of changing the code or recompiling it, but simply by specifying in an XML configuration file which sequence of handlers is needed to provide a given service. Custom handlers are used by the AXML peer to provide Web services and execute any required pre or post processing, as will be illustrated in the next section.

Architecture

We now give a high level overview of the AXML peer architecture. Figure 6.5 illustrates its main components. These can be divided into two categories: those responsible for the client-side functionalities of the AXML peer (on the left) and those responsible for its server-side ones (on the right). Components that play a role in both appear in the middle of the Figure.

[2] Available from the Java Community Process, at `http://www.jcp.org`.

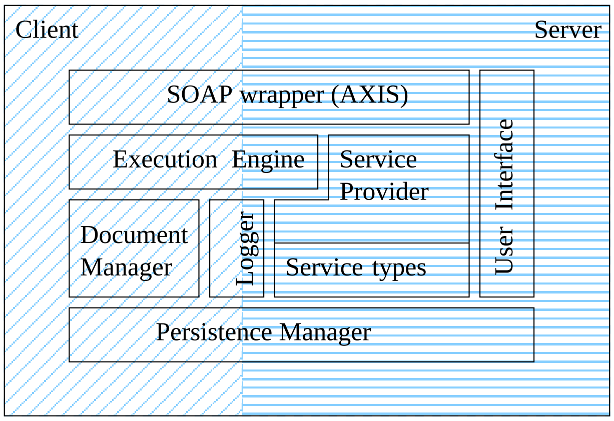

Figure 6.5: The AXML peer high level architecture

The AXML Document Manager is the module responsible for managing AXML documents. Each AXML document is internally represented as a DOM tree, which allows for easy manipulation and straightforward updates, e.g. when one of its embedded service calls is activated. Since most operations on an AXML document have to do with its embedded service calls, the latter are accessed through a hashtable, that uses unique, randomly generated, persistent identifiers as an access key.

Parameters of service calls may consist of arbitrary AXML data, and are recursively represented as AXML document objects. Recall from Section 6.1 that parameters can also be specified as relative XPath queries, which are just syntactic sugar for calls to services defined as queries. To speed up computation, we directly represent such parameters as query objects.

This module also interprets the policies specified in service calls that are related to their activation and to the validity of the data they bring. To do so, events corresponding to these policies are registered in the scheduler of the Execution Engine (see below). Additionally, the document manager is in charge of merging the results of service call invocations, following merge policies defined as in Section 6.1.

The execution engine is in charge of issuing all the service calls in the AXML peer. These can either be calls to remote Web services, or to locally defined ones, in which case the corresponding programs are invoked directly in Java, avoiding unnecessary (and expensive) SOAP communications.

Since Web services often take quite a long time to respond (generally over half a second), calling them in a blocking mode can be a serious performance bottleneck for the AXML peer. To avoid that, the execution engine needs to be able to call several services in parallel, i.e. in a non-blocking mode. To achieve this goal, the execution engine is multi-threaded. It uses a different thread for each service invocations. Thread management in this context is fairly complex, since some service calls may depend on others (e.g. for calls in parameters of other calls) and therefore some threads need to wait for the completion of others before being started.

Moreover, some service calls need not be executed immediately, but at some fixed future time, and possibly repeatedly. This is the case for instance on the client-side, for service calls embedded in documents of the AXML peer that follow a time-based explicit pull strategy. But it is also the case on the server-side, for continuous services defined by the peer that need to be reevaluated periodically. To accomplish that, the execution engine relies on a scheduler, where such operations are registered. When these tasks are due, threads are created to handle them and their execution interleaves with the tasks currently being carried out by the execution engine.

The AXML Service Provider provides the server functionality of an AXML peer. It relies on Axis for all SOAP communications and is implemented as a set of handlers that can be plugged in the Axis processing chain.

AXML focuses on *declaratively* specified services. Therefore, Axis was extended to support such specifications. We defined different service *types*, one for each declarative language we needed to support. Each service specification is an XML document that references an existing service type, defines the input and output parameters of the service, and contains its declarative specification (e.g. an X-OQL query for a service of type X-OQL). Service descriptions are stored in the persistent repository, alongside AXML documents (just like stored procedures are stored in the database in relational systems).

To implement service types, we introduced a flexible mechanism, that allows one to easily extend the AXML peer with new types of services. Each service type implementation must implement a limited set of interfaces:

- `IAXMLServiceFactory`: a factory that generates, from the declarative specification of the service, an `IAXMLService` object.
- `IAXMLService`: An object that provides an `invoke` method, to handle Web service requests.

Additionally, we provide default implementations of these interfaces, that factorize common functionality (e.g. AXML parameter handling, partial generation of the WSDL description of the service), and can be extended by each service type implementation. The service types currently available in the system are the following:

- X-OQL services: Extend X-OQL queries into parameterized queries that are able to bind some of the query variables to values passed as parameters, before the query execution.
- Update services: Extend X-OQL with simple update capabilities.
- XSLT services: simply apply an XSLT transformation on their input parameter using the Xalan processor.
- Composed services: Rely on the IBM implementation of BPEL4WS to define services as compositions of other services.
- Continuous services: periodically reevaluate a non-continuous service, and send the results as a stream of answers.

On top of providing the core functionality of a service, we saw previously that AXML often requires the use of further processing, like the lazy evaluation techniques of Chapter 5, or the schema-based safe rewriting algorithm of Chapter 4. While the implementation of each of these modules was discussed in the corresponding chapters, we focus here on how they are integrated in the AXML peer. This is done by exposing them as Axis handlers, and incorporating them in the chain of processing. Figure 6.6 shows such a chain for a query service.

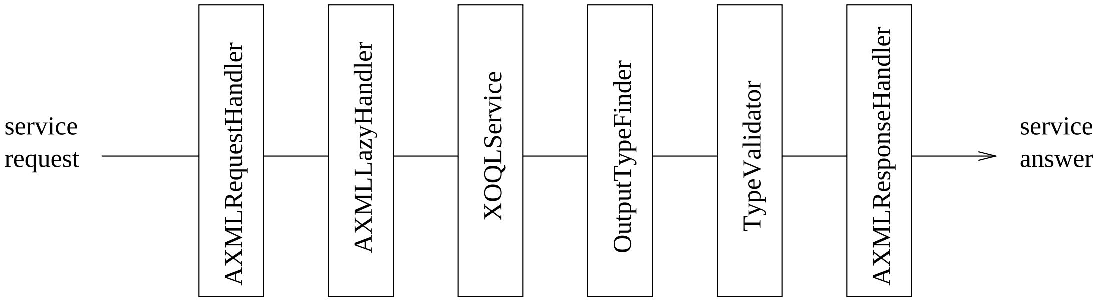

Figure 6.6: Chain of handlers for a query service

The `AXMLRequestHandler` creates AXML documents for each parameter of the service invocation, and puts them in the shared message context. It also constructs the `XOQLService` handler corresponding to the service to be answered, or retrieves it from a cache if it has been accessed recently, and *dynamically* inserts it in the chain of handlers. Then, the `AXMLLazyHandler` performs the lazy evaluation of Chapter 5, to invoke any lazy services that are relevant to answering the query, both in the documents of the repository and in the parameters of the invocation. The `XOQLService` evaluates the query corresponding to the service, and puts the result in the message context. Then, the `OutputTypeFinder` sets the target output type of the service invocation, by checking if one of the input parameters was a desired output type, and if not, by using the output type specified in the WSDL definition of the service. Eventually, the `TypeValidator` transforms, if possible, the result of the query into an instance of the type previously set, using the algorithms presented in Chapter 4, and the `AXMLResponseHandler` generates a SOAP response from the resulting AXML document.

It is important to note that chains of handlers are also used on the client side by Axis. Therefore the same handlers can be reused. For instance, the `TypeValidator` handler is used to ensure, when calling a service, that the input parameters match the type specified in its WSDL description.

The persistence manager is responsible of managing persistent information in the AXML peer. This consists of the AXML documents, the AXML service definitions, and the F-Guide summaries of documents used by the algorithms of Chapter 5.

The current version of the persistence manager is based on the filesystem. Each of the items above corresponds to an XML file stored on disk. The persistence manager is responsible for keeping the disk copy in sync with one stored in memory whenever a document, service definition, or F-guide is updated.

Work is currently going on in the AXML project to make the persistence manager support more efficient kinds of storage, mainly focusing on the Xyleme native XML database. It would also be desirable to support alternative native XML stores, especially Open Source ones, such as XXL [114] or the recent XML extension to Berkeley DB [22].

User Interface While Web services are essentially meant for communications between machines, applications are generally meant for users. The AXML peer builds on the capabilities of the Tomcat engine to expose relevant informations to users.

First, each AXML peer provides a management interface in the form of a Web site, that allows a user to view and modify the documents of the peer, to activate the service calls they contain, and also to view, modify and test the definitions of services provided by the peer.

Second, for developing custom applications, the AXML peer provides some generic JSP templates to generate a Web site from the AXML documents of the application, where embedded service calls can be invoked through HTML links (or forms, if some parameters of the calls need to be provided by the user). The application developer simply needs to write XSLT transformations to define how the data contained in the AXML documents should be presented to the user.

The Logger is a simple module that collects logs of events that happen in the AXML peer. It builds on the standard Java infrastructure for logging, and therefore provides categorizations of events into multiple levels, and exports its information into an XML stream.

The logger is mainly used for analyzing errors. In particular, it tracks the failures of invoked Web services (Time outs, or SOAP faults), and errors that may occur while processing service requests (e.g., failure to transform a service result into an instance of the output type).

6.4 Applications

We conclude this chapter by mentioning some applications we used to validate the approach. All applications presented below were built using the features of the AXML peer we just presented, and consist of relatively small sets of AXML documents, and declaratively specified AXML services. The ease of development of each of these applications shows that AXML is a powerful framework for the fast development of distributed, data-centric applications.

Peer-to-peer auctions This is the first application we built using AXML. It was demonstrated in [6]. It consists of a decentralized auctioning scenario, where each peer has some items to sell, that other peers can bid on. Each peer knows about some other peers, and can search for items they provide. The returned answers consist both of extensional data, items matching the query, and intensional data, in the form of service calls to get more items from other peers the requester didn't know about originally. Each item returned by a query answer also contains a service call that can be used to bid on the item. Eventually, when an auction closes, the winner of the auction is notified.

The main goal of this application is to illustrate the flexible discovery mechanism of new peers and auctions, that is made possible by the use of intensional answers. The application also emphasizes the possibility to perform actions on these auctions, again through service calls that are shipped with the answers.

RSS news syndication This second application was demonstrated in [2]. It further shows the benefit of exchanging AXML data, through a scenario based on the distributed syndication and querying of news headlines. A number of news sources (including actual news Web sites, such as the New York Times, or Yahoo! news) expose their latest headlines in a standard format called RSS [88]. AXML peers act as aggregators that integrate news from several sources and/or from other aggregators. Queries asked to one peer recursively propagate to the sources/aggregators it knows.

This application shows how integration services provided by peers can be customized, to control how intensional/extensional their answers are. This is done declaratively by specifying schemas in their signatures, or passing them dynamically as parameters. The algorithms presented in Chapter 4 are used to automatically match the specified schemas. The application also demonstrates continuous services, corresponding to news subscriptions, that are aware of the location of the user, again using intensional information controlled through types.

E.dot: A dynamic warehouse on food risk E.dot is a French (RNTL) project that aims to build a dynamic warehouse on food risk management, dynamically enriched with data from the Web. E.dot uses the AXML peer as a platform for the warehouse definition, construction and maintenance. It defines a high level warehouse model, which can be specified visually through a dedicated GUI. This model is then compiled into a set of AXML documents and AXML services, which are then maintained by the AXML peer. The warehouse also relies on a set of external Web services, such as the Google WebAPI, a Web crawler, a classification and clustering engine, and a PageRank module.

Electronic patient record management The goal of this application, which was demonstrated in [1], is to show that AXML can seamlessly manage distributed data and the privacy of this data, through the example of sensitive medical information. This is done by combining the AXML language with

GUPster [89], a framework developed at Lucent Technologies that unifies the description of data sources and access control requirements.

In this application, the lazy evaluation technique of Chapter 5 is at the heart of the involved computations. It is used to seamlessly push queries in a uniform way, both to relevant data sources and to Web services in charge of protecting information (by restricting the queries that can be asked).

Swan: dynamic network configuration The goal of this starting research collaboration with Alcatel, INRIA Rennes, and France Telecom, is to leverage the dynamicity of AXML in the field of network configuration. The idea is that software and hardware components will get information on how to configure or reconfigure themselves in the form of AXML data, where parts that are likely to change will be provided intensionally, in order to seamlessly account for these changes. The use of both push and pull modes of invocation is envisioned, and AXML data should also prove useful to make components dynamically aware of their environment.

6.5 Conclusion

We illustrated in this chapter the peer-to-peer architecture we proposed for the AXML language, through the features of its main component, the AXML peer. We overviewed the technical implementation of the AXML peer, that we tried to make as modular and extensible as possible, and showed how the algorithms introduced in Chapters 4 and 5 are integrated in this implementation. Work is ongoing in many directions in the AXML project, to extend this implementation, in order to provide new features or to make it more scalable. We also briefly described some applications based on AXML peers. Some of these were example scenarios, built for demonstrations, to illustrate the capabilities of the language and its implementation.

Chapter 7

Conclusion

The relational model was an important breakthrough for data management in centralized information systems. It brought a cleanly formalized data model, with strong logical foundations. The SQL query language was essential for the adoption of this model, since it gave a syntax to define queries. With the advent of the Web, new data management paradigms need to be considered which take advantage of what essentially is becoming an inherently distributed planet-scale information system.

Semistructured data, and its standard incarnation XML, are being recognized as the suitable model and language for data representation and exchange on the Web. XQuery, the query language for XML standardized by the W3C, is often advertised as the future "SQL" of the Web. However, XQuery is just one element of the puzzle, since it only allows to ask queries on centralized collections of documents. It does not indeed capture the distributed essence of the Web. For the moment, this distribution aspect is still addressed programmatically, e.g., using Java or PHP.

In this thesis, we proposed a first step towards a suitable data model and language for Web data management. Our main contribution essentially consisted in introducing intensional portions in semistructured documents, and enabling the exchange of this new kind of data. This led to a recursive scheme, that can be viewed as an analog of deductive database languages, for the Web. More precisely, we proposed Active XML, a model for distributed data management, based on XML and Web services. In this model, AXML documents can integrate information from other Web sources, through embedded calls to Web services. AXML services open new perspectives for dynamic collaboration among systems on the Web, by enabling the exchange of AXML data.

We then showed how to control this exchange of AXML data, by using declarative schema specifications where intensional information is a first class citizen, and proposing algorithms to statically enforce such extended schemas on the exchanged information. We also introduced techniques to efficiently leverage AXML data when answering queries, by taking advantage of the service calls it contains that may bring relevant information. Finally, we implemented a system that supports the AXML model, by managing persistent AXML documents and providing AXML services on top of them. This system coherently integrates the novel typing and lazy evaluation techniques we developed for the AXML model, and is built to collaborate in a distributed, peer-to-peer fashion. We plan to release the AXML peer as Open Source software, hoping that by using it, Web developers and application designers will be convinced of the usefulness of the AXML model and language, and will help it mature.

Other works around Active XML were not presented in this thesis. In [5], we introduced a formal model for AXML computations, for a restricted class of documents and services. Namely, services are assumed to be monotone, and documents are viewed as unordered trees, that are monotonously enriched by the results of their embedded service calls, and are considered modulo redundant information. We provide a well-defined semantics for documents and queries, based on a non-deterministic fixpoint computation, and obtain theoretical results on the confluence of computations, termination and lazy query evaluation, under the considered restrictions. The restrictions make the model substantially different from the one we presented here, and this is why we chose not to present it in this thesis.

Also, an extension of AXML to support distributed and replicated documents was proposed in [7]. The authors provide a cost model for query evaluation on distributed/replicated AXML documents, and show how it applies to user queries and service calls. They also describe an algorithm that, for a given peer, chooses data and services that it should replicate to improve the efficiency of maintaining and querying its AXML documents. This work was also implemented as an extension of the AXML peer, and demonstrated in [4], through a distributed workspace management scenario.

Back to our analogy, in the relational world, SQL itself is not sufficient to manage relational data. Indeed, integrity constraints, transaction management, concurrency control, access control mechanisms, or triggers are among the key peripheral yet essential functionalities offered by a relational DBMS. Also, efficient optimization techniques are required to deliver good performance.

Similar issues need also be considered in the Active XML context, to make it useful for a broad class of distributed applications. Existing techniques and results from previous works in the relational setting may be applicable, bearing in mind that AXML adds three main difficulties to cope with: (i) the flexibility of semistructured data, (ii) the distributed nature of data and computation, and (iii) the fact that a complete knowledge of the system can hardly be assumed.

Active XML opens up an array of challenging and exciting issues: transaction management for AXML services, concurrency control with fine-grained locking on AXML data, workflow management, stream management, user interface. We plan to work on some of them, and hope others will be taken over by interested researchers in the data management community. To conclude, we mention two new topics we started investigating.

We are currently considering the study of access control and security models for AXML. These are crucial, since every embedded service call can be viewed as a potential security threat. Interestingly, if simple Web services for public key encryption/decryption are available, then AXML can already express interesting security patterns such as delegation of trust, again through the exchange of intensional information. Clearly, more work is needed in that promising direction. Techniques for analyzing information flow for type-driven computations [21] may prove useful in this context.

Finally, we view the lazy evaluation technique presented here as a first step towards optimizing queries in the context of distributed intensional information. Pushing queries to capable services is a useful optimization that should be explored further, and many other interesting optimizations are possible, such as semi-join based techniques, or sideway information passing. It would be desirable to define a distributed execution model that allows to express and combine such optimizations, where peers are able to make their own optimization choices, based on both their policy and the choices of their neighbors, that leads to a globally optimized system.

Appendix A

List of Publications

Journal papers and book chapters

- **Exchanging Intensional XML Data** *Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, Fred Dang Ngoc* ACM Transactions on Database Systems (TODS), March 2005 (to appear).

- **Active XML: A Data Centric Perspective on Web Services** *Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, Roger Weber* Book chapter in Web Dynamics, Mark Levene and Alexandra Poulovassilis editors, Springer-Verlag, 2004.

Refereed papers in international conferences

- **Positive Active XML** *Serge Abiteboul, Omar Benjelloun, Tova Milo*, ACM Symposium on Principles of Database Systems (PODS), Paris, 2004.

- **Lazy Query Evaluation for Active XML** *Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, Nicoleta Preda*, ACM SIGMOD International Conference on Management of Data, Paris, 2004.

- **Exchanging Intensional XML Data** *Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, Frederic Dang Ngoc*, ACM SIGMOD International Conference on Management of Data, San Diego, 2003.

Refereed demonstrations in international conferences

- **An Electronic Patient Record on Steroids : Distributed, Peer to Peer, Secure and Privacy Conscious** *Serge Abiteboul, Bogdan Alexe, Omar Benjelloun, Bogdan Cautis, Irini Fundulaki, Tova Milo, Arnaud Sahuguet*, International Conference on Very Large Data Bases (VLDB), Toronto, 2004.

- **Schema-driven Customization of Web Services** *Serge Abiteboul, Bernd Aman, Jerome Baumgarten, Omar Benjelloun, Frederic Dang Ngoc, Tova Milo*, International Conference on Very Large Data Bases (VLDB), Berlin, 2003.

- **Active XML: Peer-to-Peer Data and Web Services Integration** *Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo and Roger Weber*, International Conference on Very Large Data Bases (VLDB), Hong Kong, 2002.

Refereed papers in national conferences

- **Active XML: A Data Centric Perspective on Web Services** *Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, Roger Weber* Journées Bases de Données Avancées, Evry, 2002.

Other publications

- **Active XML and Active Query Answers** *Serge Abiteboul, Omar Benjelloun, Tova Milo*, International Conference on Flexible Query Answering Systems, Lyon, 2004.

- **Web Services and Data Integration** *Serge Abiteboul, Omar Benjelloun, Tova Milo*, International Conference on Web Information Systems Engineering (WISE), Singapore, 2002.

- **Towards a Flexible Model for Data and Web Services Integration** *Serge Abiteboul, Omar Benjelloun, Tova Milo*, International Workshop on Foundations of Models for Information Integration, Viterbo, 2001.

Annexe B

Introduction en français

La gestion de données est sans doute l'un des aspects les plus importants de l'informatique. Depuis les années soixante, la communauté des bases de données a développé les techniques nécessaires à la gestion de données dans des entrepôts centralisés. Très tôt, de nombreux efforts ont été dévolus à l'extension de ces techniques pour également prendre en compte des données distribuées, en particulier pour leur intégration [48, 103, 94].

Mais l'avènement du Web est en train de changer la donne. Gérer des données distribuées sur le Web constitue un changement de contexte important, d'une part en raison de la grande *hétérogénéité* des sources de données et du problème posé par leur interopérabilité, et d'autre part parce que ces sources de données sont *autonomes*, et réparties à un grande échelle, celle du Web. *L'objectif de cette thèse est de proposer une approche novatrice pour intégrer données et services, qui prend en compte ces deux aspects, et de ce fait est adaptée à la gestion de données à l'échelle du Web.*

Hétérogénéité XML [109], en tant que *standard* définissant un modèle de données et un format de représentation pour les données semi-structurées, a suscité un intérêt considérable au sein de la communauté de gestion de données (par exemple, [40, 67, 82]), en raison principalement de la flexibilité qu'il apporte pour résoudre les problèmes d'hétérogénéité. Cependant, sur le Web, une grande partie des problèmes d'interopérabilité proviennent du fait que les sources exposent leur données sous forme de pages HTML, mêlant ainsi structure et présentation. De plus ces données ne sont souvent que partiellement accessibles, au travers de formulaires HTML. Heureusement, de nouvelles technologies et standards de communication sont récemment apparus, qui permettent la publication de données sur le Web et l'accès à elles de manière adéquate : les *services Web*. Ces derniers sont simplement des programmes qui acceptent en paramètres et retournent en résultat des données XML. Les standards WSDL [104] et SOAP [92] régissent respectivement la manière dont ils sont décrits et invoqués à distance à travers l'Internet. *Notre modèle mets en oeuvre efficacement XML et les services Web pour des tâches de gestion de données distribuée.*

Autonomie D'une certaine manière, les architectures centralisées d'intégration de données vont à l'encontre de l'esprit du Web, celui d'un "faible couplage" entre les systèmes participants, où le contrôle est le plus souvent partagé et les systèmes indépendants les uns des autres. De plus, le passage à l'échelle nécessaire pour prendre en charge le large volume du Web est difficile pour les systèmes centralisés. En revanche, les architectures *pair-à-pair*, basées sur un partage des ressources par échanges directs entre systèmes participants, constituent une alternative crédible aux systèmes centralisés, et ont déjà fait leurs preuves dans le contexte du partage de fichiers sur Internet [78, 45, 42]. Ces architectures capturent la nature autonome des systèmes, et leur capacité à jouer le rôle à la fois de producteurs d'information (c'est à dire de serveurs) et de consommateurs d'informations produites par d'autres (c'est à dire de clients). *Dans notre modèle, les architectures pair-à-pair sont à la base de la gestion de données distribuées.*

Nous proposons Active XML (AXML, en abbrégé), un langage qui tire profit des services Web pour la gestion de données, et qui est mis en oeuvre dans une architecture pair-à-pair. Ce langage introduit la notion de *documents AXML*, qui sont des documents XML contenant des appels à des services Web. Le modèle AXML définit également les *services AXML*, comme étant des service Web capables d'échanger des documents AXML. L'introduction de ces services conduit à des mécanismes évolués d'interaction entre systèmes, qui ont la particularité d'être centrés sur les données. Ainsi, plusieurs systèmes peuvent collaborer dynamiquement à la réalisation d'une tâche de gestion de données, et sont en particulier capables de découvrir de nouvelles sources de données pertinentes au cours de leur exécution.

Après un rappel des notions nécessaires en Chapitre 2, le modèle AXML sera présenté et motivé en Chapitre 3, ainsi qu'un état de l'art. Il est important de noter que l'inclusion d'appels d'opérations dans des données n'est pas en soi une idée nouvelle. Elle a même déjà été considérée dans le contexte de données XML et de services Web. Par exemple, dans Microsoft Office XP, les documents Office peuvent contenir des "SmartTags" pointant vers des services de la plateforme .NET [87]. Cependant, à notre connaissance, nos travaux sont les premiers à transformer des appels de services Web inclus dans des documents XML en un puissant outil de gestion de données distribuées. En combinant de manière transparente données *extensionnelles* (exprimées en XML) et *intensionnelles* (les appels de services, qui sont un moyen d'obtenir des données), notre approche capture notamment différents styles d'intégration de données, tels que ceux traditionnellement fournis par les médiateurs ou par les entrepôts de données, tout en permettant de les combiner de manière flexible.

La gestion de documents et de services AXML génère de nouvelles questions, dont l'étude est importante pour pleinement tirer profit de la flexibilité de ce modèle :

– Les données intensionnelles sont précieuses. Par conséquent, les systèmes échangeant de tels documents doivent être en mesure de contrôler de manière précise quelles parties de ces données doivent être rendues extensionnelles avant que l'échange n'intervienne. Notons que ce choix peut être influencé par diverses considérations, notamment de performances ou de sécurité. Nous proposons une solution basée sur l'utilisation de schémas, en définissant une extension adéquate de XML Schema [110], qui permet de spécifier la nature des données échangées. Nous introduisons des algorithmes utilisant des automates réguliers, qui permettent de rendre les documents échangés conformes aux schémas d'échange spécifiés. Cette partie de notre travail fera l'objet du Chapitre 4.

– Lorsqu'une requête doit être évaluée sur un document AXML, il est important de déterminer quels appels de services sont susceptibles de fournir des données pertinentes pour la réponse à la requête. Le problème est complexe, car ces appels de services peuvent être inclus n'importe où dans le document. De plus les appels de services peuvent retourner des données qui contiennent elles-mêmes de nouveaux appels de services. Nous proposons des algorithmes pour résoudre ce problème, qui sont basés sur une réécriture de la requête initiale. Cette évaluation "paresseuse" de requêtes sur des documents AXML sera présentée au Chapitre 5 de cette thèse.

Les documents et services AXML s'intègrent parfaitement à une architecture pair-à-pair, dans laquelle chaque système joue le rôle de client, en invoquant les appels de services inclus dans ses documents AXML, et celui de serveur en fournissant des services utilisant ses documents. En nous basant sur ce paradigme, nous avons implanté un système (dénommé pair AXML), qui est dédié à la gestion de documents et de services AXML. Les fonctionnalités des pairs AXML, à la fois en tant que clients et serveurs, ainsi que leur implantation, seront les sujets abordés au Chapitre 6. Les applications développées pour valider cette approche seront également présentées.

Le dernier chapitre sera notre conclusion. Nous tenterons d'y mettre nos travaux en perspective et mentionnerons quelques questions importantes qui, à notre sens, méritent d'être étudiées.

Annexe C

Conclusion en français

Le modèle relationnel a constitué une étape importante pour la gestion de données dans les systèmes d'information centralisés. Il a apporté un modèle mathématique aux fondements logiques solides. Le langage de requêtes SQL a été essentiel à l'adoption de ce modèle, en lui apportant une syntaxe simple pour exprimer des requêtes. Avec l'avènement du Web, de nouveaux paradigmes de gestion de données sont envisagés, pour tirer profit de ce qui est en train de devenir un système d'information distribué à l'échelle de la planète.

Alors que les données semi-structurées, et leur incarnation la plus populaire, XML, s'imposent comme modèle et langage de choix pour la représentation et l'échange de données sur le Web, XQuery, le langage de requêtes pour XML standardisé par le W3C, est souvent qualifié de futur SQL du Web. Mais XQuery n'est qu'une pièce du puzzle, car il permet uniquement de poser des requêtes sur des collections centralisées de documents, et ne capture donc pas la nature essentiellement distribuée du Web. Pour l'instant, cette dernière est toujours prise en charge par des programmes dédiés, écrits en Java ou en PHP par exemple.

Dans cette thèse, nous espérons avoir accompli un premier pas vers un modèle de données et un langage adéquats pour la gestion de données sur le Web. Notre contribution essentielle a consisté à introduire des portions intensionnelles dans des documents semi-structurés, et à permettre l'échange de ce nouveau type de données. Ceci nous à conduit à un langage récursif, qui peut-être vu comme un analogue des langages de bases de données déductives, pour le Web. Plus précisément, nous avons proposé Active XML, un modèle pour la gestion de données distribuées, basé sur XML et les services Web. Dans ce modèle, les documents AXML permettent l'intégration de données provenant de sources multiples, au moyen d'appels de services Web. Les services AXML quant à eux ouvrent de nouvelles perspectives de collaboration dynamique entre systèmes sur le Web, grâce à l'échange de données AXML qu'ils permettent.

Nous avons ensuite montré comment cet échange de données AXML pouvait être contrôlé, au moyen de spécifications déclaratives de schémas dans lesquels les appels de services sont des citoyens de première classe, et proposé des algorithmes permettant de garantir de manière statique la conformité des données échangées à ces schémas étendus. Nous avons également introduit des techniques permettant de tirer pleinement avantage des documents AXML lors de l'évaluation de requêtes, en profitant de ceux parmi les appels de services qu'ils contiennent qui peuvent potentiellement contribuer au résultat de la requête. Enfin, nous avons implanté un système mettant en oeuvre le modèle AXML, capable de gérer des documents AXML persistants et de fournir des services AXML les utilisant. Ce système intègre de manière cohérente les techniques nouvelles de typage et d'évaluation paresseuse de requêtes que nous avons développées pour le modèle AXML, et a été conçu pour des interactions pair-à-pair. Nous avons l'intention de diffuser le pair AXML sous forme de logiciel libre (open source), en espérant qu'à travers son utilisation, les développeurs Web et les concepteurs d'applications distribuées seront convaincus de l'utilité du modèle AXML, et contribueront à faire parvenir ce langage à maturité.

D'autres travaux autour d'Active XML n'ont pas été présentés dans cette thèse. Dans [5], nous avons introduit un modèle formel pour des classes restreintes de documents et de services AXML. Les services considérés sont des fonctions monotones, tandis que les documents sont considérés comme des arbres non ordonnés, enrichis par les invocations de leurs appels de services inclus de manière monotone également, et sont considéré modulo leurs informations redondantes. Nous fournissons une sémantique bien définie pour les documents et les requêtes dans ce contexte, basée sur un calcul de point fixe non déterministe, et obtenons des résultats théoriques sur la confluence des calculs, leur terminaison et l'évaluation paresseuse de requêtes, sous les restrictions considérées. Ces dernières rendent le modèle significativement différent de celui présenté dans cette thèse, et c'est pourquoi nous avons choisi de ne pas présenter cette partie de nos travaux.

Par ailleurs, une extension d'AXML introduisant des documents distribués et répliqués a été proposée en [7]. Les auteurs fournissent un modèle de coût pour l'évaluation de requêtes sur des document AXML distribués/répliqués et montrent comment il s'applique aux requêtes d'utilisateurs et aux appels de services. Ils décrivent également un algorithme qui, pour un pair donné, donne des directives de réplication de données et de services qui améliorent l'évaluation de requêtes et la maintenance des documents AXML du pair. Ce travail a également été implanté comme une extension du pair AXML démontrée [4] à travers un scénario de gestion d'espaces de travail distribués.

Pour en revenir à notre analogie, notons que dans le modèle relationnel, le langage SQL ne suffit pas à lui seul à gérer une base de données relationnelle. Les contraintes d'intégrité, la gestion des transactions et de la concurrence d'accès, ou encore les triggers, sont autant de fonctionnalités certes périphériques mais néanmoins essentielles qui sont fournies par un SGBD relationnel. En outre, des techniques d'optimisation efficaces sont la clé de bonnes performances.

Des aspects similaires doivent aussi être considérés dans le cadre d'Active XML, afin d'en faire un langage et un système utile pour une large classe d'applications distribuées. Des techniques et résultats existants provenant de travaux antérieurs accomplis dans le cadre du modèle relationnel peuvent certainement être transposés, à condition de prendre en compte les trois difficultés inhérentes à AXML que sont : (i) la flexibilité des données semi-structurées, (ii) la nature distribuée des données et des calculs, et (iii) le fait qu'une connaissance complète du système ne peut que difficilement être supposée.

Ainsi, Active XML conduit à un ensemble de problèmes intéressants et non triviaux, tels que la gestion de transactions pour les services AXML, la concurrence d'accès avec des verrous à grain fin pour les services AXML, la gestion de workflow, de flux continus de données, ou encore les problèmes d'interface utilisateur. Nous avons l'intention de poursuivre nos travaux dans certaines de ces directions, et espérons que ces problèmes susciteront de l'intérêt chez d'autres chercheurs.

Pour conclure, nous souhaitons mentionner deux sujets que nous avons récemment commencé d'étudier. Tout d'abord, nous nous intéressons actuellement à l'étude de modèles de sécurité et de contrôle d'accès pour AXML. Cette question est cruciale, car chaque appel de service inclus dans un document AXML peut être vu comme un risque de sécurité potentiel. Si l'on suppose que de simples services Web d'encryption/decryption à clé publique sont disponibles, nous avons observé qu'AXML permet déjà d'exprimer des protocoles de sécurité intéressants, tels que la délégation de droits. Clairement, cette direction de recherche prometteuse nécessite une étude plus approfondie. Des techniques d'analyse des flux d'information pour des évaluations dirigées par les types telles que celle présentée en [21] peuvent se révéler utiles dans ce contexte.

Par ailleurs, nous considérons que la technique d'évaluation paresseuse de requêtes présentée dans cette thèse n'est qu'un premier pas vers l'optimisation de requêtes en présence d'informations intensionnelles distribuées. Le fait de pousser des requêtes vers les services qui le permettent est une optimisation intéressante qui mérite d'être explorée plus profondément. En outre, bien d'autre techniques d'optimisation peuvent être employées, telles que celles basées sur les semi-jointures, ou le passage latéral d'informations (sideways information passing). Il serait utile de définir un modèle d'exécution distribué, permettant d'exprimer et de combiner de telles optimisations, et dans lequel les pairs participants sont à même de faire leurs propres choix d'optimisation, en fonction de leurs priorités et de celles de leurs voisins, et de manière à ce que le système global soit optimisé.

Bibliography

[1] S. Abiteboul, B. Alexe, O. Benjelloun, B. Cautis, I. Fundulaki, T. Milo, and A. Sahuguet. An Electronic Patient Record on Steroids : Distributed, Peer to Peer, Secure and Privacy Conscious. In *Proc. of VLDB*, 2004.

[2] S. Abiteboul, B. Amann, J. Baumgarten, Omar Benjelloun, Fred Dang Ngoc, and Tova Milo. Schema-driven Customization of Web Services. In *Proc. of VLDB*, 2003.

[3] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. In *Proc. of VLDB*, 1999.

[4] S. Abiteboul, J. Baumgarten, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing Distributed Workspaces with Active XML. In *Proc. of VLDB*, 2003.

[5] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *Proc. of ACM PODS*, 2004.

[6] S. Abiteboul, O. Benjelloun, T. Milo, I. Manolescu, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration (demo). In *Proc. of VLDB*, 2002.

[7] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of ACM SIGMOD*, 2003.

[8] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[9] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[10] S. Abiteboul and T. Milo. Web Services meet Datalog. Technical report, INRIA, 2004.

[11] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Int. Journal on Digital Libraries*, 1(1):68–88, April 1997.

[12] V. Aguilera. The X-OQL homepage. `http://www-rocq.inria.fr/ aguilera/xoql`.

[13] V. Aguilera. *Querying XML Documents*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 2002.

[14] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying XML Sources Using an Ontology-Based Mediator. In *Proc. of CoopIS*, 2002.

[15] The Apache Axis soap Engine. `http://xml.apache.org/axis`.

[16] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.

[17] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of ACM PODS*, pages 1–15, 1986.

[18] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for XML. In *Proc. of ACM SIGMOD*, 2002.

[19] H. P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 321–363. Elsevier, Amsterdam, 1990.

[20] C. Beeri, A. Y. Levy, and M.-C. Rousset. Rewriting queries using views in description logics. In *Proc. of ACM PODS*, 1997.

[21] V. Benzaken, M. Burelle, and G. Castagna. Information Flow Security for XML Transformations. In *Proc. Of ASIAN*, 2003.

[22] Berkeley DB XML. Available from `http://www.sleepycat.com/`.

[23] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. of ICDE*, 2002.

[24] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *Proc. of the Int. WWW Conf.*, Hong Kong, China, May 2001.

[25] Business Process Execution Language for Web Services Version 1.1. Available from
`http://www.ibm.com/developerworks/library/ws-bpel/`.

[26] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proc. of ICDE*, 2003.

[27] P. Buneman, S. B. Davidson, M. F. Fernandez, and D Suciu. Adding structure to unstructured data. In *Proc. of ICDT*, 1997.

[28] K. Selcuk Candan, Sushil Jajodia, and V. S. Subrahmanian. Secure Mediated Databases. In *Proc. of ICDE*, pages 28–37, 1996.

[29] L. Cardelli. Abstractions for Mobile Computation. In *Secure Internet Programming*, pages 51–94, 1999.

[30] L. Cardelli and A. D. Gordon. Mobile Ambients. In M. Nivat, editor, *Proc. of FoSSaCS*, volume 1378, pages 140–155. Springer-Verlag, Berlin, Germany, 1998.

[31] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1994.

[32] V. Christophides, R. Hull, A. Kumar, and J. Siméon. Workflow Mediation using VorteXML. *IEEE Data Engineering Bulletin*, 24(1):40–45, March 2001.

[33] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an adaptive path index for XML. In *Proc. of ACM SIGMOD*, 2002.

[34] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proc. of ACM SIGMOD*, pages 177–188, 1998.

[35] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proc. of ICDE*, 2002.

[36] C. Cremarenco. Implementation of the Active XML Peer for the J2ME platform, 2003. Internship report, available from `http://purl.org/net/axml`.

[37] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing XML Documents. In *Proc. of EDBT*, 2001.

[38] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proc. of ACM SIGMOD*, pages 509–520. ACM Press, 2001.

[39] The Document Object Model (DOM).
`http://www.w3.org/DOM/`.

[40] D. Draper, A. Y. Halevy, and D. S. Weld. The Nimble XML Data Integration System. In *Proc. of ICDE*, pages 155–160, 2001.

[41] H. V. Jagadish et al. TIMBER: A native XML database. *The VLDB Journal*, 11(4), 2002.

[42] The Freenet Project.
`http://freenetproject.org`.

[43] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8:117–132, 1997.

[44] G. Gardarin, A. Mensch, and A. Tomasic. An Introduction to the e-XML Data Integration Suite. In *Proc. of EDBT*, 2002.

[45] The Gnutella homepage. `http://www.gnutella.com`.

[46] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, 1997.

[47] G. Gottlob and C. Koch. Efficient algorithms for processing XPath queries. In *Proc. of VLDB*, 2002.

[48] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases*. IEEE Press, 1989.

[49] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proc. of ICDT*, pages 98–112, 1997.

[50] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[51] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proc. of the Int. WWW Conf.*, 2003.

[52] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. on Programming Languages and Systems*, 7(4):510–538, 1985.

[53] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[54] H. Hosoya and B. C. Pierce. “XDuce: A Typed XML Processing Language”. In *Proc. of WebDB*, Dallas, TX, May 2000.

[55] The HyperText Markup Language (HTML 4.01).
`http://www.w3.org/TR/html4`.

[56] J. Mc Hugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. Technical report, Stanford University Database Group, Feb 1997.

[57] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *Proc. of ACM PODS*, 2003.

[58] International Organization for Standardization. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, August 1986.

[59] The SUN Java Technology.
`http://java.sun.com/`.

[60] Jelly: Executable XML.
`http://jakarta.apache.org/commons/sandbox/jelly`.

[61] T. Jim and D. Suciu. Dynamically Distributed Query Evaluation. In *Proc. of ACM PODS*, pages 413–424, 2001.

[62] SUN’s Java Server Pages.
`http://java.sun.com/products/jsp/`.

[63] The Kazaa Homepage.
`http://www.kazaa.com`.

[64] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proc. of VLDB*, pages 251–262, 1996.

[65] Macromedia Coldfusion MX.
`http://www.macromedia.com/software/coldfusion/`.

[66] M. Makoto. Relax (regular language description for xml). ISO/IEC Technical Report, may 2001.

[67] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proc. of VLDB*, 2001.

[68] Microsoft and IBM. The WS-Security specification.
`http://www.ibm.com/webservices/library/ ws-secure/`.

[69] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. of ACM PODS*, 2002.

[70] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Dang Ngoc. Exchanging intensional XML data. In *Proc. of ACM SIGMOD*, 2003.

[71] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proc. of ACM PODS*, 1999.

[72] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. of ACM PODS*, pages 11–22, 2000.

[73] J. C. Mitchell. Type Systems for Programming Languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 365–458. Elsevier, Amsterdam, 1990.

[74] H.G. Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[75] The Morpheus homepage.
`http://www.morpheus-os.com`.

[76] A. Muscholl, T. Schwentick, and L. Segoufin. Active Context-Free Games. In *Proc. of STACS*, 2004.

[77] Namespaces in XML.
`http://www.w3.org/TR/REC-xml-names`.

[78] The Napster homepage.
`http://www.napster.com`.

[79] F. Dang Ngoc. Validation de documents XML contenant des appels de services. Master's thesis, CNAM, 2002. DEA SIR (in French).

[80] Oracle XML Developer's Kit for Java.
`http://otn.oracle.com/tech/xml/`.

[81] Web ontology language.
`http://www.w3.org/2004/OWL/`.

[82] Ozone: Integrating Structured and Semistructured Data. T. Lahiri and S. Abiteboul and J. Widom. In *Proc. Int. Workshop on Database Programming Languages*, 1999.

[83] Y. Papakonstantinou, V. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, and P. Velikhov. Xml queries and algebra in the enosys integration platform. *Data Knowl. Eng.*, 44(3):299–322, 2003.

[84] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proc. of ACM SIGMOD*, 1999.

[85] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Query set specification language (QSSL). In *Proc. of WebDB*, 2003.

[86] The PHP Hypertext Preprocessor.
`http://www.php.net`.

[87] J. Powell and T. Maxwell. Integrating Office XP Smart Tags with the Microsoft .NET Platform.
`http://msdn.microsoft.com`.

[88] RSS 1.0 Specification.
`http://purl.org/rss/1.0`.

[89] A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong. Enter Once, Share Everywhere: User Profile Management in Converged Networks. In *Proc. of CIDR*, 2003.

[90] The Simple API for XML (SAX).
`http://sax.sourceforge.net`.

[91] L. Segoufin. Personal Communication, 2003.

[92] Simple Object Access Protocol (SOAP) 1.1.
`http://www.w3.org/TR/SOAP`.

[93] M. Stonebraker, E. Anderson, E. Hanson, and B. Rubenstein. QUEL as a data type. In *Proc. of ACM SIGMOD*, 1984.

[94] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition*. Prentice-Hall, 1999.

[95] I. Tatarinov, Z. Ives, A. Levy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

[96] The Apache Tomcat Servlet Container.
`http://jakarta.apache.org/tomcat`.

[97] Universal Description, Discovery, and Integration of Business for the Web (UDDI).
`http://www.uddi.org`.

[98] J.D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[99] L. Vieille. Recursive axioms in deductive databases: The Query/Subquery approach. In L. Kerschberg, editor, *Proc. First Intl. Conf. on Expert Database Systems*, pages 179–193, 1986.

[100] The World Wide Web Consortium.
`http://www.w3.org/`.

[101] G. Weikum, editor. *Infrastructure for Advanced E-Services*, volume 24, no. 1. Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society edition, Mar 2001.

[102] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann Publishers, 1996.

[103] G. Wiederhold. Intelligent Integration of Information. In *Proc. of ACM SIGMOD*, pages 434–437, Washington, DC, May 1993.

[104] Web Services Definition Language (WSDL). `http://www.w3.org/TR/wsdl`.

[105] Web Services Flow Language (WSFL 1.0). Available from `http://www.ibm.com/`.

[106] The Xerces Java Parser. `http://xml.apache.org/xerces-j/`.

[107] The Extensible HyperText Markup Language (XHTML 1.0). `http://www.w3.org/TR/xhtml1`.

[108] XLANG, Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.

[109] The Extensible Markup Language (XML) 1.0 (2nd Edition). `http://www.w3.org/TR/REC-xml`.

[110] The XML Schema specification. `http://www.w3.org/TR/XML/Schema`.

[111] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.

[112] XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`.

[113] XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`.

[114] The XXL Project.
`http://www.mathematik.uni-marburg.de/dbs/xxl`.

[115] Xyleme Home Page.
`http://www.xyleme.com`.

[116] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *The VLDB Journal*, pages 136–145, 1997.

[117] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of ACM SIGMOD*, pages 316–327, 1995.