Exchanging Intensional XML Data

TOVA MILO
INRIA and Tel-Aviv University
SERGE ABITEBOUL
INRIA
BERND AMANN
Cedric-CNAM and INRIA-Futurs
and
OMAR BENJELLOUN and FRED DANG NGOC
INRIA

XML is becoming the universal format for data exchange between applications. Recently, the emergence of Web services as standard means of publishing and accessing data on the Web introduced a new class of XML documents, which we call *intensional* documents. These are XML documents where some of the data is given explicitly while other parts are defined only intensionally by means of embedded calls to Web services.

When such documents are exchanged between applications, one has the choice of whether or not to materialize the intensional data (i.e., to invoke the embedded calls) before the document is sent. This choice may be influenced by various parameters, such as performance and security considerations. This article addresses the problem of guiding this materialization process.

We argue that—like for regular XML data—schemas (à la DTD and XML Schema) can be used to control the exchange of intensional data and, in particular, to determine which data should be materialized before sending a document, and which should not. We formalize the problem and provide algorithms to solve it. We also present an implementation that complies with real-life standards for XML data, schemas, and Web services, and is used in the Active XML system. We illustrate the usefulness of this approach through a real-life application for peer-to-peer news exchange.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms: Algorithms, Languages, Verification

Additional Key Words and Phrases: Data exchange, intensional information, typing, Web services, XML

This work was partially supported by EU IST project DBGlobe (IST 2001-32645).
This work was done while T. Milo, O. Benjelloun, and F. D. Ngoc were at INRIA-Futurs.
Authors' current addresses: T. Milo, School of Computer Science, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel; email: milo@cs.tau.ac.il; S. Abiteboul and B. Amann, INRIA-Futurs, Parc Club Orsay-University, 4 Rue Jean Monod, 91893 Orsay Cedex, France; email: {serge,abiteboul, bernd.amann}@inria.fr; O. Benjelloun, Gates Hall 4A, Room 433, Stanford University, Stanford, CA 94305-9040; email: benjelloun@db.stanford.edu; F. D. Ngoc, France Telecom R&D and LRI, 38–40, rue du Général Leclerc, 92794 Issy-Les Moulineaux, France; email: Frederic.dangngoc@rd.francetelecom.com.

1. INTRODUCTION

XML, a self-describing semistructured data model, is becoming the standard format for data exchange between applications. Recently, the use of XML documents where some parts of the data are given explicitly, while others consist of programs that generate data, started gaining popularity. We refer to such documents as *intensional documents*, since some of their data are defined by programs. We term *materialization* the process of evaluating *some of the programs* included in an intensional XML document and replacing them by their results. The goal of this article is to study the new issues raised by the *exchange* of such intensional XML documents between applications, and, in particular, how to decide which parts of the data should be materialized before the document is sent and which should not.

This work was developed in the context of the Active XML system [Abiteboul et al. 2002, 2003b] (also see the Active XML homepage of Web site `http://www-rocq.inria.fr/verso/Gemo/Projects/axml`). The latter is centered around the notion of *Active XML documents*, which are XML documents where parts of the content is explicit XML data whereas other parts are generated by calls to Web services. In the present article, we are only concerned with certain aspects of Active XML that are also relevant to many other systems. Therefore, we use the more general term of *intensional* documents to denote documents with such features.

To understand the problem, let us first highlight an essential difference between the exchange of regular XML data and that of intensional XML data. In frameworks such as those of Sun[1] or PHP,[2] intensional data is provided by programming constructs embedded inside documents. Upon request, *all the code* is evaluated and replaced by its result to obtain a regular, fully materialized HTML or XML document, which is then sent. In other terms, only *extensional* data is exchanged. This simple scenario has recently changed due to the emergence of standards for *Web services* such as SOAP, WSDL,[3] and UDDI.[4] Web services are becoming the standard means to access, describe and advertise valuable, dynamic, up-to-date sources of information over the Web. Recent frameworks such as Active XML, but also Macromedia MX[5] and Apache Jelly[6] started allowing for the definition of intensional data, by embedding calls to Web services inside documents.

This new generation of intensional documents have a property that we view here as crucial: since Web services can essentially be called from everywhere on the Web, one does not need to materialize all the intensional data before sending a document. Instead, a more flexible data exchange paradigm is possible, where the sender sends an intensional document, and gives the receiver the freedom

[1]See Sun's Java server pages (JSP) online at `http://java.sun.com/products/jsp`.
[2]See the PHP hypertext preprocessor at `http://www.php.net`.
[3]See the W3C Web services activity at `http://www.w3.org/2002/ws`.
[4]UDDI stands for Universal Description, Discovery, and Integration of Business for the Web. Go online to `http://www.uddi.org`.
[5]Macromedia Coldfusion MX. Go online to `http://www.macromedia.com/`.
[6]Jelly: Executable xml. Go online to `http://jakarta.apache.org/commons/sandbox/jelly`.

to materialize the data if and when needed. In general, one can use a hybrid approach, where some data is materialized by the sender before the document is sent, and some by the receiver.

As a simple example, consider an intensional document for the Web page of a local newspaper. It may contain some extensional XML data, such as its name, address, and some general information about the newspaper, and some intensional fragments, for example, one for the current temperature in the city, obtained from a weather forecast Web service, and a list of current art exhibits, obtained, say, from the *TimeOut* local guide. In the traditional setting, upon request, all calls would be activated, and the resulting fully materialized document would be sent to the client. We allow for more flexible scenarios, where the newspaper reader could also receive a (smaller) intensional document, or one where some of the data is materialized (e.g., the art exhibits) and some is left intensional (e.g., the temperature). A benefit that can be seen immediately is that the user is now able to get the weather forecast whenever she pleases, just by activating the corresponding service call, without having to reload the whole newspaper document.

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 whether or not to materialize some intensional data:

—*Performance*. The decision of 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, for example, 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, for example, because of the 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, for example, for a UDDI-like service registry, the origin of the information is what is truly requested by the receiver, and hence service

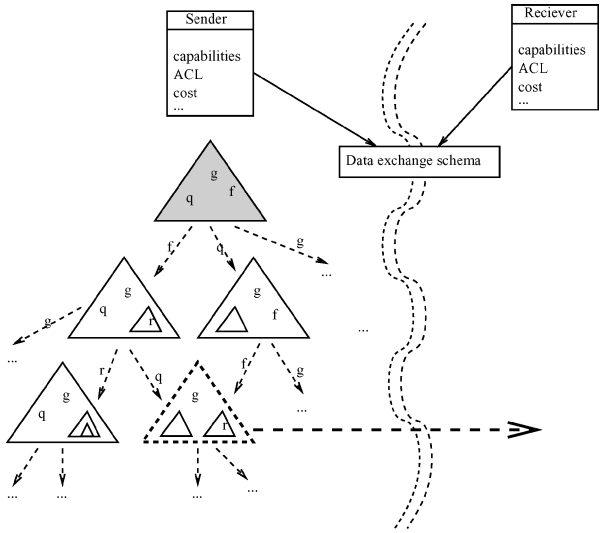

Fig. 1. Data exchange scenario for intensional documents.

calls should not be materialized. In other cases, one may prefer to hide the true origin of the information, for example, 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 10 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 structure, matches the schema expected by the receiver. If not, the sender must perform the required calls for transforming the data into the desired structure, if this is possible.

A typical such scenario is depicted in Figure 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 them, the sender must find at least one document conforming 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, which are specified in their WSDL descriptions. The techniques presented in this article can be used to achieve that.

The contributions of the article are as follows:

(1) We provide a simple but flexible XML-based syntax to embed service calls in XML documents, and introduce an extension of XML Schema 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, for example, a *temperature* element, and accepting a service call returning some data of this type, for example, $() \rightarrow$ *temperature*.

(2) 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.

(3) 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.

(4) We illustrate the flexibility of the proposed paradigm through a real-life application: peer-to-peer news syndication. We will show that Web services can be *customized* by using and enforcing several exchange schemas.

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 those provided by Cluet et al. [1998] and Doan et al. [2001]. 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. For presentation reasons, 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 in this article are related to standard typing problems in programming languages [Mitchell 1990], they differ here due to the regular expressions present in XML schemas. Indeed, the general problem that will be formalized here was recently shown to be undecidable by Muscholl et al. [2004]. We will 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 Active XML system [Abiteboul et al. 2002] (see also the Active XML homepage of Web site `http://www-rocq.inria.fr/verso/Gemo/Projects/axml`). This system provides persistent storage for intensional documents with embedded calls to Web services, along with active features to automatically trigger these services and thus enrich/update the intensional documents. Furthermore, it allows developers to declaratively specify Web services that support intensional documents as input and output parameters. We used the algorithms described here to implement a module that controls the types of documents being sent to (and returned by) these Web services. This module is in charge of materializing the appropriate data fragments to meet the interface requirements.

In the following, we assume that the reader is familiar with XML and its typing languages (DTD or XML Schema). Although some basic knowledge about SOAP and WSDL might be helpful to understand the details of the implementation, it is not necessary.

The article is organized as follows: Section 2 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 3 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 and 5, respectively. The problem of checking compatibility between intensional schemas is considered in Section 6. The implementation is described in Section 7. Then, we present in Section 8 an application of the algorithms to Web services customization, in the context of peer-to-peer news syndication. The last section studies related works and concludes the article.

2. THE MODEL AND THE PROBLEM

To simplify the presentation, we start by formalizing the problem using a simple data model and 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 intensional 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.

2.1 The Simple Model

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

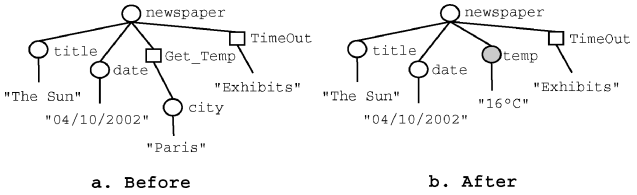

Fig. 2. An intensional document before/after a call.

2.1.1 *Simple Intensional XML Documents.* We model intensional XML 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,[7] 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 2.1. An intensional 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 the *function parameters*. When the function is called, these subtrees are passed to it. The return value then replaces the function node in the document. This is illustrated in Figure 2, where data nodes are represented by circles, function nodes are represented by squares, and data values are quoted. Here, the *Get_Temp* Web service is invoked with the city name as a parameter. It returns a *temp* element, which replaces the function node. An example of the actual XML representation of intensional documents is given in Section 7. Observe that the parameter subtrees and the return values may themselves be intensional documents, that is, contain function nodes.

2.1.2 *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 2.2. A document schema s 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 $\tau_{out}(f)$.

[7]We assume in this model that function names identify Web service operations. This translates in the implementation to several parameters (URL, operation name, ...) that allow one to invoke the Web services.

For instance, the following is an example of a schema:

$$
\begin{array}{ll}
 & \textbf{data}: \\
 & \tau(newspaper) = title.date.(Get\_Temp \mid temp).(TimeOut \mid exhibit^*) \\
 & \tau(title) = data \\
 & \tau(date) = data \\
 & \tau(temp) = data \\
 & \tau(city) = data \\
 & \tau(exhibit) = title.(Get\_Date \mid date) \\
(*) & \textbf{functions}: \\
 & \tau_{in}(Get\_Temp) = city \\
 & \tau_{out}(Get\_Temp) = temp \\
 & \tau_{in}(TimeOut) = data \\
 & \tau_{out}(TimeOut) = (exhibit \mid performance)^* \\
 & \tau_{in}(Get\_Date) = title \\
 & \tau_{out}(Get\_Date) = date
\end{array}
$$

We next define the semantics of a schema, that is, the set of its instances. To do so, if R is a regular expression over $L \cup F$, 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 2.3. An intensional document t is an *instance* of a schema $s = (L, F, \tau)$ if for each data node (respectively function node) $n \in t$ with label $l \in L$ (respectively $l \in F$), the labels of n's children form a word in $lang(\tau(l))$ (respectively in $lang(\tau_{in}(l))$).

For a function name $f \in F$, a sequence $t_1, \ldots, t_n$ of intensional trees is an *input instance* (respectively *output instance*) of f, if the labels of the roots form a word in $lang(\tau_{in}(f))$ (respectively $lang(\tau_{out}(f))$, and all the trees are instances[8] of s.

It is easy to see that the document of Figure 2(a) is an instance of the schema of $(*)$, but not of a schema with τ' identical to τ above, except for

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

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

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

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 of $(***)$. However, it is *possible* that it becomes one (if

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

$TimeOut$ returns a sequence of *exhibits*). The only way to know is to call the service.

This type of "on-line" testing is fine 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 nevertheless.

2.1.3 *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.

Definition 2.4. For a tree 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.[9] If $t \xrightarrow{v_1} t_1 \xrightarrow{v_2} t_2 \cdots \xrightarrow{v_n} t_n$ 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 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 2.5. Let t be a tree and s 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 such that 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 2(a) *safely rewrites* into the schema of $(**)$ but only *possibly rewrites* into that of $(***)$.

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 structure required by the second application (e.g., the agreed data exchange format).

Definition 2.6. 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'.

For instance, consider the schema of $(*)$ presented above with *newspaper* as the root label. This schema safely rewrites into the schema of $(**)$ but does not safely rewrite into the one of $(***)$.

[9]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).

2.1.4 *The Results.* Going back to the data exchange scenario described in the introduction, we can now specify our main contributions:

(1) 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, and
(2) When safe rewriting is not possible, we present an algorithm that tests whether t may be possibly rewritten into s, and finds a possibly successful rewriting sequence, if one exists.
(3) We also provide an algorithm for testing, given two schemas, whether one can be safely rewritten into the other.

2.2 A Richer Data Model

In order to make our presentation clear, and to simplify the definition of document and schema rewritings, we used 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.

2.2.1 *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 2(a), for different cities. A common intensional schema for such documents should not require the use of a particular *Get_temp* 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, for example, 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 belongs to 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 one that requires that the function signature only be *subsumed* by the one specified in the definition, that is, that every instance of the former be also an instance of the latter. This is possible but is computationally more heavy, since it entails checking inclusion of the tree language defined by the two schemas.

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

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

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.

2.2.2 *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*.[10] 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.[11] Therefore, we can use wildcards to allow certain document parts to contain arbitrary sub-trees with arbitrary functions, or restrict them to (respectively 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.

2.2.3 *Restricted Service Invocations.* Another interesting extension is the following: 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 *noninvocable* ones. A *legal* rewriting is then one that invokes only invocable functions. The notions of safe and possible

[10]The W3C XML activity. Go online to `www.w3.org/XML`.
[11]Which is described in its WSDL description and, in our model, is one of the components of the function name.

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.

2.2.4 *XML, XML Schema, and WSDL.* The simple XML trees considered above ignore a number of features of XML, such as attributes, and use a single domain for data values. A richer setting may be obtained by using the full fledged XML data model (see footnote 10). Similarly, richer schemas may be defined by adopting XML Schema (see footnote 10), rather than using the simple DTD-like schema 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. 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 7.

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 7. 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.

3. EXCHANGING INTENSIONAL DATA

We start by considering document rewriting. Schema rewriting is considered later in Section 6.

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 (preferably the shortest or cheapest one, according to some criteria).
(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 (preferably with the least side effects on the path to find it, and at the least cost).

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.

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 7 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.

3.1 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, reexamining 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[12] 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 the real-life examples that we considered, left-to-right rewritings were not limiting.

3.2 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 ("*") subexpressions, the 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, other function calls. In general the number of such new functions may be unbounded. For instance, consider a *Get_Exhibits*

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

function, with output type

$$\tau_{out}(Get\_Exhibits) = Get\_Exhibit^*.$$

When *Get_Exhibits* is invoked, an arbitrarily large number of *Get_Exhibit* functions may be returned, and one has 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.

3.3 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 3.1. For a rewriting sequence $t \xrightarrow{v_1} t_1 \cdots \xrightarrow{v_n} t_n$, 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 nodes contains no paths of length greater than k.

The restriction. The restriction that we will impose below is the following: We will consider only k-depth left-to-right rewritings.

Note that while this restriction limits the search space, the latter remains infinite, due to the starred subexpressions 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 most practical cases. The problem of arbitrary safe rewriting (without the left-to-right k-depth restriction) was recently shown to be undecidable [Muscholl et al. 2004]. Further work by the same authors [Muscholl et al. 2004; Segoufin 2003] has shown 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)—and thus are mostly of theoretical interest.

4. 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 signature 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 subelements. 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,

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, is handled. In particular, we show how to decide which of the functions among these children needs 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 7.

4.1 Rewriting Function Parameters

To invoke a function, its parameters should be of the right type. If they are not, 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, that is, 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 outmost function calls in t are fine. We can thus ignore their parameters (and whatever functions that appear in them) and need to safely rewrite t into s by invoking only these outmost calls.

4.2 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 outmost functions have already been handled, and we need to safely rewrite the tree (forest) by invoking only these outmost functions. To do that we can traverse the tree(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 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 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.[13] 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 Hopcroft and Ullman [1979].

[13]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)
1	Build finite state automata for the following regular languages:
2	(a) An automaton A_w accepting w as a single word.
3	(b) Automata A_{f_i}, $i = 1 \dots n$, each accepting the regular language R_{f_i}.
4	(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.
5	Let $A_w^k := A_w$.
6	For $j = 1, \dots, k$
7	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:
8	(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.
9	(b) Denote v as a *fork* node (for the edge e).
10	(c) The two *fork options* of v (for e) are e itself and the new outgoing ϵ edge.
11	Construct the Cartesian product automaton $A_\times = A_w^k \times \overline{A}$.
12	The *fork nodes* and *fork options* in $A_\times$ reflect those of A_w^k:
13	(a) the *fork nodes* $[q, p] \in A_\times$ are those where q was a fork node in A_w^k.
14	(b) Similarly, a *fork option* in $A_\times$ consists of all edges originating from one fork option edge in A_w^k.
15	Mark nodes in $A_\times$ as follows.
16	(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.
17	(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.
18	Return **true** if the initial state is marked, **false** otherwise.

Fig. 3. Safe rewriting of w into R.

Given the word w, the output types $R_{f_1}, \dots, R_{f_n}$ of the available functions, and the target regular language R, the algorithm in Figure 3 tests if w can be safely rewritten into a word in R. Then, if the answer is positive, the algorithm presented in Section 4.4 finds a safe rewriting sequence.

We give the intuition behind the first algorithm next. To illustrate, we use the newspaper document in Figure 2(a). Assume that we look at the root *newspaper* node. Its children labels form the word $w = \mathit{title.date.Get\_Temp.TimeOut}$. Assume that we want to find a safe rewriting for this word into a word in the regular language $\tau'(\mathit{newspaper})$ of the schema of (**), namely,

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

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.

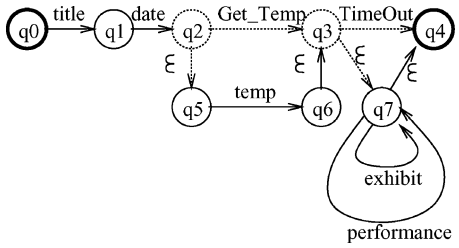

Fig. 4. The A_w^1 automaton from the newspaper document.

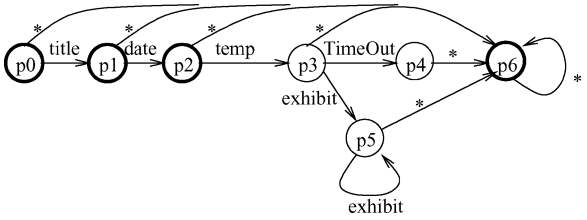

Fig. 5. The complement automaton $\overline{A}$ for schema (**).

The automaton A_w^k 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 shows the 1-depth automaton A_w^1 for the word w = *title.date.Get_Temp.TimeOut*, with the signature of the *Get_Temp* and *TimeOut* functions defined as in Section 2. q_2 and q_3 are the fork nodes and their two outgoing edges represent their fork options for *Get_Temp* and *TimeOut*, respectively. An ϵ edge represents the choice of invoking the function while a function edge represents the choice not to invoke it.

Suppose first that we want to verify that all possible rewritings lead to a "good" word, that is, that they belong to the target language R. To put things in 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 of (**) is given in Figure 5. 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 and 5, respectively) is given in Figure 6. The initial state is $[q_0, p_0]$ and the final accepting one is $[q_4, p_6]$.

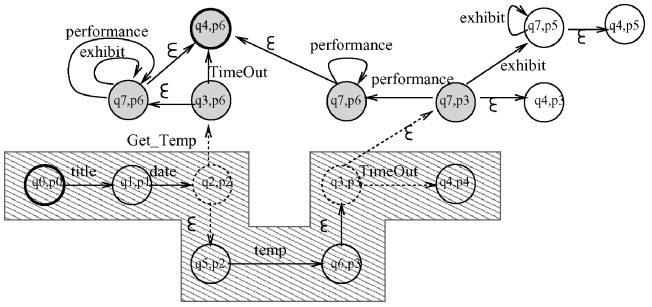

Fig. 6. The Cartesian product automaton $A_\times$.

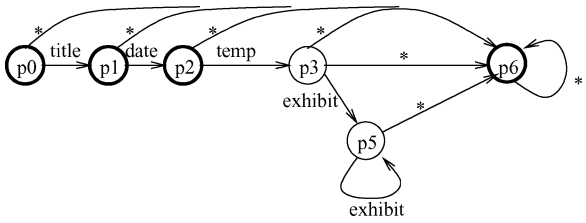

Fig. 7. The complement automaton $\overline{A'}$ for schema (***).

Note, however, that, when searching for a safe rewriting, one does not need to verify that all possible rewritings lead to a "good word," that is, that *none* of 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 6. 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 of (**). We will see in Section 4.4 how to find this rewriting.

For another example, consider the schema of (***). 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 7. To test whether it is possible to safely rewrite our newspaper document into this schema, we construct a Cartesian product automaton $A'_\times = A_w^1 \times \overline{A'}$ (with A_w^1 as in Figure 4 and $\overline{A'}$ as in Figure 7). $A'_\times$ is given in Figure 8.

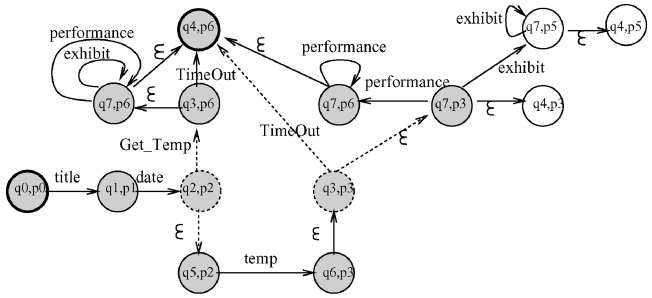

Fig. 8. The Cartesian product automaton $A'_{\times}$.

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 of (***). Note that this is precisely what our intuitive discussion from Section 2 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 4.1. *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 subautomaton "originating" from A_w, and use A^j, $0 < j \leq k$ to denote the subautomaton "originating" from some A_f added to A_k to represent the possible outputs of f, at the jth 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 jth iteration. To simplify the notation we use A^j to denote any representative of this set.

—Given a subautomaton A^j, an initial (respectively final) state of A^j is a state $[q, q']$ such that q is an initial (respectively final) state of A_w/A_{f_i}.

Completeness. We start by proving that the algorithm is complete, that is, 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 4.2. *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 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, then 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, lets look at the one that has the longest marked prefix in A^j.[14] 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's suppose it's 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 the 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 the longest 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. □

We are now ready to prove direction (i). We do this again by contradiction. Assume that our algorithm returns a negative answer, that is, that the initial state of $A_\times$ is marked, but a k-depth left-to-right safe rewriting from A_w does exists. 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 counterexample 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 chose any answer we want for a function call, as long as it matches its output type. The rewriting should succeed anyway.

[14]Note that it is not necessarily unique.

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 4.2.

At each step, when we traverse an edge with a label in L, we simply output its label. If the edge is labeled by a function name in 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 4.2, 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 makes it end on a word that is not in the target language R, a contradiction.

Soundness. We now turn to direction (ii), which states the soundness of our algorithm—namely, that if the initial state is not marked, then every word accepted by A_w can be rewritten to match the target schema. We start by proving that if the algorithm succeeds, then the following proposition holds:

PROPOSITION 4.3. *Let A^j be a subautomaton of $A_\times$ corresponding to some function automaton A_{f_i} (or to A_w if $j = 0$).*

For every nonmarked state $[q, q']$ of A^j originating from A_{f_i} (respectively A_0), every word in $L(A_{f_i}, q)$ (respectively $L(A_w, q)$) has a "safe rewriting" into a word w' such that w' corresponds to a nonmarked path in $A_\times$ leading to a (nonmarked) final state of A^j.[15]

PROOF. We use induction on j, starting from $j = k$ and going down to $j = 0$, to show that every word in $L(A_{f_i}, q)$ (respectively $L(A_w, q)$) that contains only

[15]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.

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 nonmarked, we get to a nonmarked accepting state of A^j, which means we are done. Otherwise, if the path contains marked nodes, we show, by a second induction on the number of function symbols in w, how to rewrite w to a "good" path.

The base of the induction is for a word w that doesn't contain function nodes. Then, clearly, all nodes on the corresponding path must be nonmarked, 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 subwords $w = w_1.f_i.w_2$. Since $[v, v']$ is not marked, its other fork option (the ϵ edge corresponding to f_i), must lead to a nonmarked initial state of the subautomaton 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 nonmarked 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 nonmarked path leads to a final state of A^j. It follows that the rewritten word $w_1.w'.w''$, and its corresponding nonmarked path, leads from $[q, q']$ to a nonmarked final state A^j via a path consisting only of nonmarked nodes. □

We are now ready to prove direction (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 nonmarked 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 4.1. □

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

	find safe rewriting (word w, marked automaton $A_\times$)
1	To obtain such a rewriting:
2	(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.
3	– The non marked *fork options* on the path determine the rewriting choices (i.e. which functions to call).
4	– When a function is invoked we continue the path with the new rewritten word (rather than the original w).
5	(b) To minimize the rewriting cost, chose a path with minimal number/cost of function invocations.
6	exit

Fig. 9. Finding the rewriting of w into R.

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 $O(|A_\times|^2) = O((|A_w^k| \times |\overline{A}|)^2)$. The size of A_w^k is at most $O((|s_0| + |w|)^k)$ and the size of the complement automaton $\overline{A}$ is at most exponential in the automaton being complemented [Hopcroft and Ullman 1979], namely, at most exponential in the size of the target schema s. This exponential blow up may happen however only when s uses nondeterministic regular expressions (i.e., regular expressions whose corresponding finite state automaton is nondeterministic). 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).

4.4 Finding a Rewriting

The algorithm of Figure 3 checks if a safe rewriting exists. The constructive proof we used to show its soundness entails a way to find a rewriting sequence when a safe rewriting exists, which corresponds to the algorithm of Figure 9.

This algorithm finds the safe rewriting sequence by following a nonmarked path. Each fork node on the path, together with its nonmarked 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 discribed, which corresponds to Figure 6, it is easy to see (following the path with colored background) that *Get_Temp* 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 $w \times x^k$.

4.5 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

	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 3
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.

Fig. 10. Possible rewriting of w into R.

get their actual output, and then tries to safely rewrite the document. In terms of the algorithm of Figure 3, 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.

5. 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 nonfeasible (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 functions 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 10 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 [Hopcroft and Ullman 1979].

For instance, consider the automaton A for the schema of (***) with *newspaper* structure *title.date.temp.exhibit** given in Figure 11. The initial state is p_0 and the final accepting states are p_3 and p_4. The Cartesian product automaton $A_\times = A_w^1 \times A$ (for A_k^1 as in Figure 4 and A as in Figure 11) is given in Figure 12. 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 *Get_Temp*

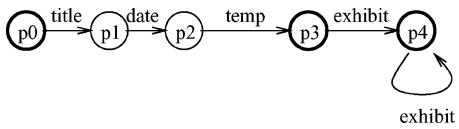

Fig. 11. An automaton A for schema (***).

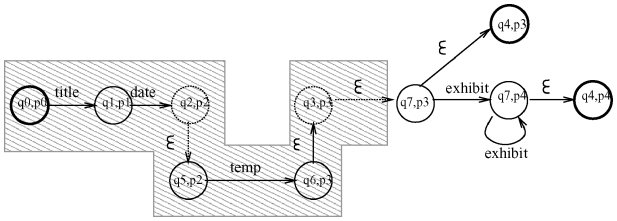

Fig. 12. Cartesian product automaton for *possible* rewriting.

	find possible rewriting (word w, marked automaton $A_\times$)
1	To obtain such a rewriting:
2	(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 3).
3	(c) Backtrack when the calls return a value that does not allow to continue to an accepting state.
4	(d) To minimize the rewriting cost, chose a path with minimal number/cost of function invocations.
5	Exit when a final state is reached, or all choice were tried.

Fig. 13. Finding a possible rewriting of w into R.

and *TimeOut* functions. If *TimeOut* returns nothing but *exhibits* the rewriting succeeds.

The correctness of this algorithm is stated below.

PROPOSITION 5.1. *The above algorithm returns true iff a k-depth possible rewriting exists.*

PROOF. Since A_w^k 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_w^k 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 13. We follow a marked path, and invoke functions 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.

6. 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 DTDs, 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.

6.1 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 that (b) 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 6.3.

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.

6.2 Rewriting XML Schemas

Things are more involved when we consider more expressive schema languages, in the style of XML Schema. Types are allowed to be decoupled from element labels, but it holds that 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 there

	language safe rewriting (initial language R_0, 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_{R_0} accepting the regular language R_0.
3	(b) Automata A_{f_i}, $i = 1 \dots n$, respectively accepting the regular languages R_{f_i}.
4	(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.
5	Let $A^k_{R_0} := A_{R_0}$.
6	For $j = 1, \dots, k$
7	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:
8	(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.
9	(b) Denote v as a *fork* node (for the edge e).
10	(c) The two *fork options* of v (for e) are e itself and the new outgoing ϵ edge.
11	Construct the Cartesian product automaton $A_\times = A^k_{R_0} \times \overline{A}$.
12	The *fork nodes* and *fork options* in $A_\times$ reflect those of $A^k_{R_0}$:
13	(a) the *fork nodes* $[q, p] \in A_\times$ are those where q was a fork node in $A^k_{R_0}$.
14	(b) Similarly, a *fork option* in $A_\times$ consists of all edges originating from one fork option edge in $A^k_{R_0}$.
15	Mark nodes in $A_\times$ as follows.
16	(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.)
17	(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.
18	Return **true** if the initial state is marked, **false** otherwise.

Fig. 14. Language safe rewriting of R_0 into R.

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 hasn't been tested yet need to be processed. This ensures that the algorithm terminates, even if schemas are recursive.

6.3 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 14 checks just that.

This algorithm is almost identical to the one presented in Section 4, 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 6.1. *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 naturally extends to language safe rewriting. There, the completeness of the algorithm was shown by building a counterexample 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 doesn't rewrite into R. The soundness of the algorithm of Section 4 was shown in a constructive manner, by building a word corresponding to a nonmarked path in $A_\times$. The same construction applies to *each* word accepted by A_{R_0}, that is, for each word in the language R_0, which establishes the correctness of this algorithm. □

7. IMPLEMENTATION

The ideas and algorithms presented in the previous sections have been implemented and used in the *Schema Enforcement* module of the Active XML system [Abiteboul et al. 2002] (also see the Active XML homepage of Web site `http://www.rocq.inria.fr/verso/Gemo/Projects/axml`). We next present how the intensional data model and schema language of the previous sections map to XML, XML Schema, SOAP, and WSDL. Then, we briefly describe the ActiveXML system and the *Schema Enforcement* module.

7.1 Using the Standards

In the implementation, an intensional XML document is a syntactically well-formed XML document. This is because we also use an XML-based syntax to express the intensional parts in it. To distinguish these parts from the rest of the document, we rely on the mechanism of XML namespaces (see footnote 10). More precisely, the namespace `http://www.activexml.com/ns/int` is defined for service calls. These calls can appear at any place where XML elements are allowed. The following example corresponds to the document of Figure 2(a):

```
<?xml version="1.0"?>
  <newspaper xmlns:int="http://www.activexml.com/ns/int">
    <title> The Sun </title>
    <date> 04/10/2002 </date>
    <int:fun endpointURL="http://www.forecast.com/soap"
             methodName="Get_Temp"
             namespaceURI="urn:xmethods-weather">
      <int:params>
        <int:param>
          <city>Paris</city>
        </int:param>
      </int:params>
    </int:fun>
    <int:fun endpointURL="http://www.timeout.com/paris"
             methodName="TimeOut">
             namespaceURI="urn:timeout-program">
```

```
      <int:params>
        <int:param> exhibits </int:param>
      </int:params>
    </int:fun>
  </newspaper>
```

Function nodes have three attributes that provide the necessary information to call a service using the SOAP protocol: the URL of the server, the method name, and the associated namespace. These attributes uniquely identify the called function, and are isomorphic to the function name in the abstract model.

In order to define schemas for intensional documents, we use XML Schema$_{int}$, which is an extension of XML Schema. 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 subelements: *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, that is, 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*, which 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 *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 (see footnote 10)—namely, when parsing a document, for each element and each function node, the subelements 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).

7.2 The ActiveXML System

ActiveXML is a peer-to-peer system that is centered around intensional XML documents. Each peer contains a repository of intensional documents, and provides some active features to enrich them by automatically triggering the function calls they contain. It also provides some Web services, defined declaratively as queries/updates on top of the repository documents. All the exchanges

between the ActiveXML peers, and with other Web service providers/consumers use the SOAP protocol.

The important point here is that both the services that an ActiveXML peer invokes and those that it provides potentially accept intensional input parameters and return intensional results. Calls to "regular" Web services should comply with the input and output types defined in their WSDL description. Similarly, when calling an ActiveXML peer, the parameters of the call should comply with its WSDL. The role of the *Schema Enforcement* module is (i) to verify whether the call parameters conform to the WSDL$_{int}$ description of the service, (ii) if not, to try to rewrite them into the required structure and (iii) if this fails, to report an error. Similarly, before an ActiveXML service returns its answer, the module performs the same three steps on the returned data.

7.3 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 [Ngoc 2002]. 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[16] and Oracle Schema Processor[17]) appeared to have limited extensibility. Our parser relies on a standard event-based SAX parser.[16] 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, and in particular the three same stages:

(1) checking function parameters recursively, starting from the most inner 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. 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. 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,

[16]The Xerces Java parser. Go online to `http://xml.apache.org/xerces-j/`.
[17]The Oracle XML developer's kit for Java. Go online to `http://otn.oracle.com/tech/xml/`.

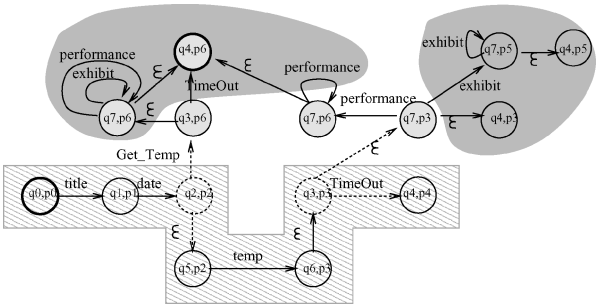

Fig. 15. The pruned automaton.

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 5 and 7). 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. For example, in Figure 15, the top left shaded area illustrates which parts of the Cartesian product automaton of Figure 6 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 3, it saves a lot of unnecessary computation in practice. Details are available in Ngoc [2002].

8. PEER-TO-PEER NEWS SYNDICATION

In this section, we will illustrate the exchange of intensional documents, and the usefulness of our schema-based rewriting techniques through a real-life application: peer-to-peer news syndication. This application was recently demonstrated in Abiteboul et al. [2003a].

The setting is the one shown on Figure 16. We consider a number of news sources (newspaper Web sites, or individual "Weblogs") that regularly publish news stories. They share this information with others in a standard XML format, called RSS.[18] Clients can *periodically* query/retrieve news from the sources they are interested in, or *subscribe* to news feeds. News *aggregators* are special peers that know of several news sources and let other clients ask queries to and/or discover the news sources they know.

[18]RSS 1.0 specification. Go online to `http://purl.org/rss/1.0`.

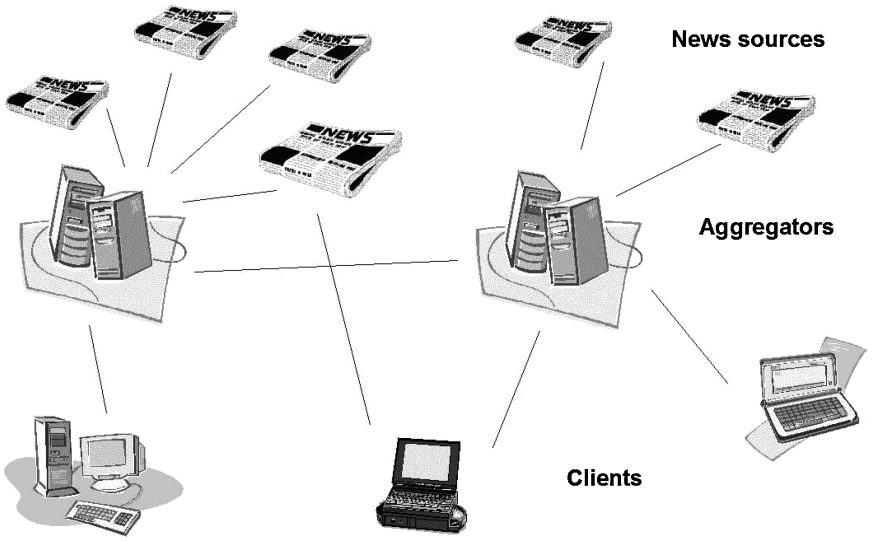

Fig. 16. Peer-to-peer news exchange.

All interactions between news sources, aggregators, and clients are done through calls to Web services they provide. Intensional documents can be exchanged both when passing parameters to these Web services, and in the answers they return. These exchanges are controlled by XML schemas, and documents are rewritten to match these schemas, using the safe/possible rewriting algorithms detailed in the previous sections.

This mechanism is used to provide *several versions* of a service, without changing its implementation, merely by using different schemas for its input parameters and results. For instance, the *same* querying service is easily customized to be used by distinct kinds of participants, for example, various client types or aggregators, with different requirements on the type of its input/output.

More specifically, for each kind of peer we consider (namely, news sources and aggregators), we propose a set of basic Web services, with intensional output and input parameters, and show how they can be customized for different clients via schema-based rewriting. We first consider the customization of intensional outputs, then the one of intensional inputs.

8.1 Customizing Intensional Outputs

News sources provide news stories, using a basic Web service named `getStory`, which retrieves a story based on its identifier, and has the following signature:

```
<function id="GetStory">
  <params>
    <param>
      <xsd:simpleType ref="xsd:string" />
    </param>
  </params>
```

```
  <result>
    <xsd:element name="story" type="xsd:string" />
  </result>
</functionPattern>
```

Note that the output of this service is fully extensional. News sources also allow users to search for news items by keywords,[19] using the following service:

```
<function id="GetNewsAbout">
  <params>
    <param>
      <xsd:simpleType ref="xsd:string" />
    </param>
  </params>
  <result>
    <xsd:complexType ref="ItemList2" />
  </result>
</functionPattern>
```

This service returns an RSS list of news items, of type `ItemList2`, where the items are given extensionally, except for the story, which can be intensional. The definition of the corresponding function pattern, `intensionalStory` is omitted.

```
<xsd:complexType name="ItemList2">
  <xsd:sequence>
    <xsd:element name="item" type="Item"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Item">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element ref="pubDate" type="xsd:dateTime"/>
    <xsd:element ref="description" type="xsd:string"/>
    <xsd:choice>
      <xsi:functionPattern ref="intensionalStory"/>
      <xsd:element name="story" type="xsd:string"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:NMTOKEN"/>
</xsd:complexType>
```

A fully extensional variant of this service, aimed for instance at PDAs that download news for offline reading, is easily provided by employing the *Schema Enforcement* module to rewrite the previous output to one that complies to a fully extensional `ItemList3` type, similar to the one above, except for the story that *has* to be extensional.

[19]More complex query languages, such as the one proposed by Edutella could also be used (go online to `http://edutella.jxta.org`).

A more complex scenario allows readers to specify a desired output type at call time, as a parameter of the service call. If there exists a rewriting of the output that matches this schema, it will be applied before sending the result, otherwise an error message will be returned.

Aggregators act as "superpeers" in the network. They know a number of news sources they can use to answer user queries. They also know other aggregators, which can relay the queries to additional news sources and other aggregators, transitively. Like news sources, they provide a `getNewsAbout` Web service, but allow for a more intensional output, of type `ItemList`, where news items can be either extensional or intensional. In the latter case they must match the `intensionalNews` function pattern, whose definition is omitted.

```
<xsd:complexType name="ItemList">
  <xsd:sequence>
    <xsd:choice>
      <xsi:functionPattern ref="intensionalNews"/>
      <xsd:element name="item" type="Item"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

When queried by simple news readers, the answer is rewritten, depending if the reader is a RSS customer or a PDA, into a `Itemlist2` or `Itemlist3` version, respectively. On the other hand, when queried by other aggregators that prefer compact intensional answers which can be easily forwarded to other aggregators, no rewriting is performed, with the answer remaining as intensional as possible, preferably complying to the type below, which *requires* the information to be intensional.

```
<xsd:complexType name="ItemList4">
  <xsd:sequence>
    <xsi:functionPattern ref="intensionalNews"/>
  </xsd:sequence>
</xsd:complexType>
```

Note also that aggregators may have different capabilities. For instance, some of them may not be able to recursively invoke the service calls they get in intensional answers. This is captured by having them supply, as an input parameter, a precise type for the answer of `getNewsAbout`, that matches their capabilities (e.g., return me only service calls that return extensional data).

8.2 Intensional Input

So far, we considered the intensional output of services. To illustrate the power of intensional input parameters, we define a *continuous* version of the `getNewsAbout` service provided by news sources and aggregators.

Clients call this service only once, to subscribe to a news feed. Then, they periodically get new information that matches their query (a dual service exists, to unsubscribe). Here, the input parameter is allowed to be given intensionally,

so that the service provider can probe it, adjusting the answer to the parameter's current value. For instance, consider a mobile user whose physical location changes, and wants to get news about the town she is visiting. The zip code of this town can be provided by a Web service running on her device, namely a GPS service. A call to this service will be passed as an intensional query parameter, and will be called by the news source in order to periodically send her the relevant local information.

This continuous news service is actually implemented using a wrapper around a noncontinuous `getNewsAbout` service, calling the latter periodically with the keyword parameter it received in the subscription. Since `getNewsAbout` doesn't accept an intensional input parameter, the *schema enforcement* module rewrites the intensional parameter given in the subscription every time it has to be called.

8.3 Demonstration Setting

To demonstrate this application [Abiteboul et al. 2003a], news sources were built as simple wrappers around RSS files provided by news websites such as Yahoo!News, BBC Word, the New York Times, and CNN. The news from these sources could also be queried through two aggregators providing the `GetNewsAbout` service, but customized with different output schemas. The customization of intensional input parameters was demonstrated using a continuous service, as explained above, by providing a call to a `getFavoriteKeyword` service as a parameter for the subscription.

9. CONCLUSION AND RELATED WORK

As mentioned in the Introduction, XML documents with embedded calls to Web services are already present in several existing products. The idea of including function calls in data is certainly not a new one. Functions embedded in data were already present in relational systems [Molina et al. 2002] as stored procedures. Also, method calls form a key component of object-oriented databases [Cattell 1996]. In the Web context, scripting languages such as PHP (see footnote 2) or JSP (see footnote 1) 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 (see footnote 17). Embedding Web service calls in XML documents is also done in popular products such as Microsoft Office (Smart Tags) and Macromedia MX.

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 toward a more precise description of XML documents embedding computation. Further work in that direction is clearly needed to better understand this powerful paradigm. There are a number of other proposals for typing XML documents, for example, Makoto [2001], Hosoya and Pierce [2000], and Cluet et al. [1998]. We selected XML Schema (see footnote 10) 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 (see footnote 3). By extending XML Schema, we naturally introduce function types/patterns in WSDL service signatures. Finally, one aspect of XML Schema simplifies the problem we study, namely, the unambiguity of XML Schema grammars.

In many applications, it is necessary to screen queries and/or results according to specific user groups [Candan et al. 1996]. 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, for example, in Damiani et al. [2001] and WS-Security.[20] However, further work is needed to investigate this aspect.

The work presented here is part of the ActiveXML [Abiteboul et al. 2002, 2003b] (see also the Active XML homepage of the Web site: `http://www.rocq.inria.fr/verso/Gemo/Projects/axml`) project based on XML and Web services. We presented in this article what forms the core of the module that, in a peer, 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 peers 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, for example, UDDI style search (see footnote 4) to try to find services on the Web that provide some particular information.

In the global ActiveXML project, research is going on to extend the framework in various directions. In particular, we are working on distribution and replication of XML data and Web services [Abiteboul et al. 2003a]. 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 [Ozsu and Valduriez 1999] and to distributed computing at large. The novel aspect is the ability to exchange intensional information. This is in spirit of Jim and Suciu [2001], which considers also the exchange of intensional information in a distributed query processing setting.

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 [Garcia-Molina et al. 1997] or to propagate changes for warehouse maintenance [Zhuge et al. 1995]. 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).

[20]The WS-Security specification. Go online to `http://www.ibm.com/webservices/library/ws-secure/`.

On the other hand, this is orthogonal to the issue of selecting the views to materialize in a warehouse, studied in, for example, Gupta [1997] and Yang et al. [1997].

To conclude, 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, many results were found by Muscholl et al. [2004]. Namely, 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 Milo et al. [2000], which are based on tree automata.

REFERENCES

ABITEBOUL, S., AMANN, B., BAUMGARTEN, J., BENJELLOUN, O., NGOC, F. D., AND MILO, T. 2003a. Schema-driven customization of Web services. In *Proceedings of VLDB*.

ABITEBOUL, S., BENJELLOUN, O., MANOLESCU, I., MILO, T., AND WEBER, R. 2002. Active XML: Peer-to-peer data and Web services integration (demo). In *Proceedings of VLDB*.

ABITEBOUL, S., BONIFATI, A., COBENA, G., MANOLESCU, I., AND MILO, T. 2003b. Dynamic XML documents with distribution and replication. In *Proceedings of ACM SIGMOD*.

CANDAN, K. S., JAJODIA, S., AND SUBRAHMANIAN, V. S. 1996. Secure mediated databases. In *Proceedings of ICDE*. 28–37.

CATTELL, R., Ed. 1996. *The Object Database Standard: ODMG-93*. Morgan Kaufman, San Francisco, CA.

CLUET, S., DELOBEL, C., SIMÉON, J., AND SMAGA, K. 1998. Your mediators need data conversion! In *Proceedings of ACM SIGMOD*. 177–188.

DAMIANI, E., DI VIMERCATI, S. D. C., PARABOSCHI, S., AND SAMARATI, P. 2001. Securing XML documents. In *Proceedings of EDBT*.

DOAN, A., DOMINGOS, P., AND HALEVY, A. Y. 2001. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proceedings of ACM SIGMOD*. ACM Press, New York, NY, 509–520.

GARCIA-MOLINA, H., PAPAKONSTANTINOU, Y., QUASS, D., RAJARAMAN, A., SAGIV, Y., ULLMAN, J., AND WIDOM, J. 1997. The TSIMMIS approach to mediation: Data models and languages. *J. Intel. Inform. Syst. 8*, 117–132.

GUPTA, H. 1997. Selection of views to materialize in a data warehouse. In *Proceedings of ICDT*. 98–112.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.

HOSOYA, H. AND PIERCE, B. C. 2000. XDuce: A typed XML processing language. In *Proceedings of WebDB* (Dallas, TX).

JIM, T. AND SUCIU, D. 2001. Dynamically distributed query evaluation. In *Proceedings of ACM PODS*. 413–424.

MAKOTO, M. 2001. RELAX (Regular Language description for XML). ISO/IEC Tech. Rep. ISO/IEC, Geneva, Switzerland.

MILO, T., SUCIU, D., AND VIANU, V. 2000. Typechecking for XML transformers. In *Proceedings of ACM PODS*. 11–22.

MITCHELL, J. C. 1990. Type systems for programming languages. In *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier, Amsterdam, The Netherlands, 365–458.

MOLINA, H., ULLMAN, J., AND WIDOM, J. 2002. *Database Systems: The Complete Book*. Prentice Hall, Englewood Cliffs, NJ.

MUSCHOLL, A., SCHWENTICK, T., AND SEGOUFIN, L. 2004. Active context-free games. In *Proceedings of the 21st Symposium on Theoretical Aspects of Computer Science* (STACS '04; Le Comm, Montpelier, France, Mar. 25–27).

NGOC, F. D. 2002. Validation de documents XML contenant des appels de services. M.S. thesis. CNAM. DEA SIR (in French) University of Paris VI, Paris, France.

OZSU, T. AND VALDURIEZ, P. 1999. *Principles of Distributed Database Systems* (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

SEGOUFIN, L. 2003. Personal communication.

YANG, J., KARLAPALEM, K., AND LI, Q. 1997. Algorithms for materialized view design in data warehousing environment. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufman Publishers, San Francisco, CA, 136–145.

ZHUGE, Y., GARCÍA-MOLINA, H., HAMMER, J., AND WIDOM, J. 1995. View maintenance in a warehousing environment. In *Proceedings of ACM SIGMOD*. 316–327.

Received October 2003; accepted March 2004