Exchanging Intensional XML Data *

Tova Milo† Serge Abiteboul Bernd Amann Omar Benjelloun Fred Dang Ngoc

INRIA Futurs
FirstName.LastName@inria.fr

ABSTRACT

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 to materialize the intensional data (i.e. to invoke the embedded calls) or not, before the document is sent. This choice may be influenced by various parameters, such as performance and security considerations. This paper addresses the problem of guiding this materialization process.

We argue that, just like for regular XML data, schemas (ala DTD and XML Schema) may 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 [3, 1].

1. INTRODUCTION

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

*This project is partially supported by EU IST project DBGlobe (IST 2001-32645)

†On sabbatical from Tel-Aviv University

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

This work was developed in the context of the Active XML system and language [3, 1, 2]. The system is centered around 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 (typically performing queries or updates). In the present paper, we are only concerned with certain aspects of Active XML, that are common to many other systems, described bellow. We will thus use the term *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 Sun's JSP [15], or php [24], 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. This simple scenario has recently changed due to emerging standards for *Web services* such as SOAP and WSDL [27], and UDDI [26], which are becoming the standard means of accessing and describing and advertising valuable, dynamic, up-to-date sources of information over the Web. Recently, frameworks such as Macromedia MX [16], Apache Jelly [13] and Active XML [3, 1, 2], allow 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 be called from essentially anywhere on the Web, one does not need anymore to materialize all the intensional data before sending the document. Instead, a more flexible data exchange paradigm is possible, where the sender sends an intensional document and gives the receiver the freedom 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 home-page of a local newspaper. It may contain some extensional XML data, such as general information about the newspaper, and some intensional fragments, e.g. one for the current temperature in the city, obtained from a weather forecast Web service, and a listing of current art exhibits, obtained from the *TimeOut* local guide. A newspaper reader may receive the full materialized document, a smaller intensional one, or one where some data is materialized (e.g. the temperature) and some left intensional (e.g. art exhibits).

Before getting to the description of the technical solution we propose, let us see some of the considerations that may guide the choice of materializing or not some information:

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

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

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

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

Observe that the data returned by a service, say *TimeOut*, may itself contain some intensional parts. Therefore the decision of whether or not to materialize some information is inherently a recursive process. For instance, for a receiver who cannot handle intensional documents, the newspaper server would have to recursively materialize all the data 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 where in the documents. Before sending information, the sender must check if the data, in its current structure, matches the schema expected by the receiver and if not, the sender must perform the required calls for transforming the data into the desired structure, if 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 gray 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.

The contributions of the paper are as follows:

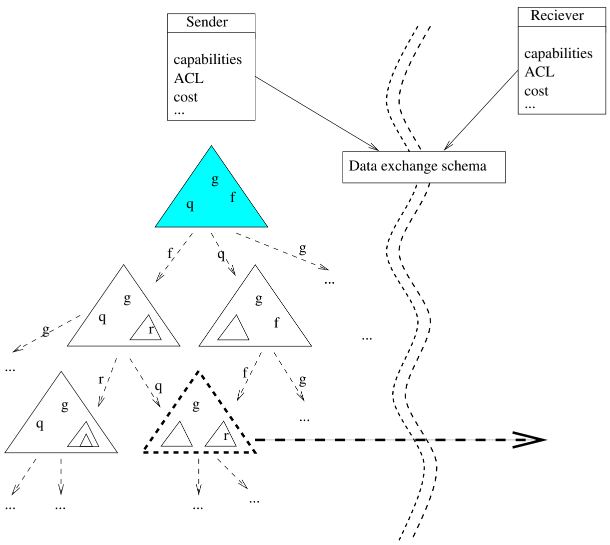

Figure 1: Data exchange scenario for intensional documents

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, e.g. a *temperature* element, and accepting a service call returning some data of this type, e.g., $() \rightarrow$ *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 guarantees 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.

As explained above, our algorithms allow to 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 [6, 8]. As a simple example, one may need to convert a temperature from Celsius degrees to Fahrenheit. Although such aspects are clearly complementary and could be added to our framework, they are not considered here. The focus is on the issue of partially materializing the *given data* to match the specified schema.

The core technique in this work is based on automata theory. For presentation reasons, we detail a simplified version of the main algorithm. We briefly sketch 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 paper are related to standard problems in programming languages typing [20], things are different here, due to the regular expressions present in XML schemas. Indeed, the general problem that will be formalized here is still open. We introduce a restriction that is practically founded and that allows users to provide a solution.

All the ideas presented here have been implemented and tested in the context the Active XML system[3, 1]. 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 users 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 received by, those Web services. The 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). Whereas basic notions of SOAP and WSDL might be helpful to understand the details, they are not necessary.

The paper 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 introduces the key restriction that we consider. It also introduces the notions of "safe" and "possible" rewriting which are studied in Section 4 and 5 resp. The problem of checking compatibility between intensional schemas is considered in Section 6. The implementation is briefly described in Section 7. The last section studies related works and concludes .

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 take into account the features ignored by the first simple model, and in particular how a richer XML Schema-style schema is employed.

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

DEFINITION 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 $\lambda : N \to \mathcal{L} \cup \mathcal{F} \cup \mathcal{D}$ 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 function nodes are represented by squares. Here, the

[1]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 to invoke the Web services.

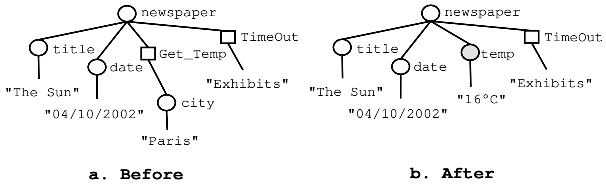

Figure 2: An intensional document before/after a call.

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 for intensional documents will be given in Section 7. Observe that the parameter subtrees and the return values may themselves be intensional documents, i.e. contain function nodes.

Simple schema. 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 the input and output types.

DEFINITION 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 $\tau_{out}(f)$.*

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

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

We next define the semantics of a schema, i.e., 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 3. *An intensional document t is an* instance *of a schema $s = (L, F, \tau)$ if for each data node (resp. function node) $n \in t$ with label $l \in L$ (resp. $l \in F$), the labels of n's children form a word in $lang(\tau(l))$ (resp. in $lang(\tau_{in}(l))$).*

For a function name $f \in F$, a sequence $t_1, \ldots, t_n$ of intensional trees is an input instance *(resp.* output instance*) of f, if the labels of the roots form a word in $lang(\tau_{in}(f))$ (resp. $lang(\tau_{out}(f))$, and all the trees are instances [2] of s.*

[2]Like in DTD's, every subtree conforms to the same schema as the whole document.

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

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

However, the document can always be turned into an instance of the schema in $(**)$, 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:

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

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

This type of "on line" testing is 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.

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 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* [3]*. If $t \xrightarrow{v_1} t_1 \xrightarrow{v_2} t_2 \ldots \xrightarrow{v_n} t_n$ we say that t* rewrites *into t_n, denoted $t \xrightarrow{*} t_n$. The nodes $v_1, \ldots, v_n$ are called the* rewriting sequence. *The set of all trees t' s.t. $t \xrightarrow{*} t'$ is denoted $ext(t)$.*

Note that in the rewriting process, the replacement of a function node v by its output instance is independent of any function semantics. In particular, we may replace two occurrences of the same function 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 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 s.t. all trees t' where $t \xrightarrow{v} t'$ rewrite safely 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 in $(**)$ 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).

[3]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).

DEFINITION 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 schema $(*)$ above with *document* as the root label. The schema safely rewrites into schema $(**)$ but does not safely rewrite into schema $(***)$.

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, nevertheless, 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.

2.1 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.

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 proper signature. The particular weather forecast function that will be used may depend on the city and be, for instance, retrieved from some UDDI service registry. We may also want to check that the function is safe according to some security policy.

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.

In terms of implementation, one can assume that the boolean predicate is implemented as a Web service that takes a function name as input and returns True/False.

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

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

This schema enforces the fact that the function used in the document has the proper signature and satisfies the boolean predicates $UDDIF$ and $InACL$. The first predicate ($UUDIF$) is a Web service that checks if the given function (service) is registered in some particular UDDI registry. Predicate $InACL$ then verifies if the client has the necessary access privileges for executing the given function (calling the service).

Wildcards. Together with function patterns, one may also use wildcards in schemas. Their use is already common for *data*, to express the fact that a certain part of a document may contain an arbitrary unconstrained subtree. XML Schema further allows to restrict wildcards to (or exclude from them) certain domains of data, through the use of namespaces [29]. This extends naturally to our context, using wildcards to allow certain document parts to contain arbitrary sub-trees with arbitrary functions, or restrict it to (resp. exclude from it) certain classes of functions.

The combination of wildcards and function patterns allows for great flexibility in describing the structure of documents. For instance, one may specify that the temperature is obtained from an arbitrary function that returns a correct *temp* element, but may take any argument, being data or function call.

Restricted service invocations. 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 *non-invocable* ones. A *legal* rewriting is then one that invokes only invocable functions. The notions of safe and possible rewritings extend naturally to consider only legal rewritings. Since we are interested here only in such rewritings, whenever we talk in the sequel about a function invocation, we mean an invocable one.

XML and XML Schema. 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 [29]. Similarly, richer schemas may be defined by adopting XML Schema [29], 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, functions 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.

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 an agreed data exchange schema s, the sender needs to rewrite t into s. A possible process is as follows:

1. Check if t rewrites safely 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 the unsafe case, 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 XMLSchema.

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 issued - things may in general be much more complex. We explain next why this is the case and present a restriction that will make the problem tractable.

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

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 the 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 function $Get\_Exhibits$ with output type

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

When the function $Get\_Exhibits$ is invoked, an arbitrary 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.

[4]Similar development can be done for a right-to-left processing

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 handles 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 7. *For a rewriting sequence $t \xrightarrow{v_1} t_1 \ldots \xrightarrow{v_n} t_n$, we say that a function node v_j depends on a function node v_i if $v_j \in t_i$ but $\notin t_{i-1}$ (namely if the node v_j was returned by the invocation of the function v_i).*

We say that a rewriting sequence is of depth k *if the dependency graph among the 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 the restriction limits the search space, the latter remains infinite due to the starred sub expressions appearing in the schema. However, under this restriction, we can exhibit a finite representation (based on automata) of the search space and automata-based techniques to solve the safe rewriting problem.

Even with this restriction, the framework is general enough to handle most practical cases. It remains open whether the problem of arbitrary safe rewriting (without the left-to-right k-depth restriction) is decidable. We proved decidability of arbitrary safe rewriting for a restricted class of schemas but those are only of theoretical interest. Due to space limitations, this will not be presented here.

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 rewriting).

To simplify, we assume that common functions have the same definitions in s_0 and s. 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 and the algorithm can be extended to handle distinct signatures. We omit this here for space reasons.

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 core idea, is used there. We will explain the main principles of this variant in Section 7.

Rewriting function parameters. In order to be able 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, namely 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 outer-most function calls in t are fine. We can thus ignore their parameters (and whatever functions appear in them) and need to safely rewrite t into s by invoking only these outer-most calls.

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 out-most functions have already been handled, and we need to safely rewrite the tree (forest) by invoking only these out-most 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 [5]. We can thus 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

[5]Note that since we assumed that s_0 and s agree on function signatures, we only need to rewrite the *original* children of n and not those that were returned by function invocation. Without this assumption these will have to be rewritten as well.

those invocations. To conclude the description of our rewriting algorithm we thus only need to explain how this is done.

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 [6]. 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 will use in our algorithm standard notions such as the intersection and complement of regular languages and the cartesian product of automata. For some basic material see for instance [11].

Given the word w, the output types $R_{f_1}, \ldots, R_{f_n}$ of the available functions, and the target regular language R, the algorithm in Figure 3 tests if w can be safely rewritten into a word in R, and if so finds a safe rewriting sequence.

	safe rewriting (word w, functions output types $R_{f_1} \ldots R_{f_n}$, 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}, $i = 1 \ldots n$, 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 $j = 1, \ldots, k$
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	**A safe rewriting exists iff the initial state is not marked.**
19	To obtain such a rewriting:
20	(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.
21	– The non marked *fork options* on the path determine the rewriting choices (i.e. which functions to call).
22	– When a function is invoked we continue the path with the new rewritten word (rather than the original w).
23	(b) To minimize the rewriting cost, chose a path with minimal number/cost of function invocations.
24	exit

Figure 3: Safe rewriting of w into R.

We give the intuition behind this 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 $w = title.date.Get\_Temp.TimeOut$. Assume that we want to find a safe rewriting for this word into a word in the regular language $\tau'(newspaper)$ of the schema in (**), namely

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

The process of rewriting involves choosing some functions in w and replacing them by a possible output; then choosing some other functions (which might have been returned by the previous calls) and replacing them by their output, and so on, up to depth k. For each function occurrence we have two choices: either to leave it untouched, or to replace it by some word in it output type. The automaton A_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 $TimeOut$, resp. 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 rewriting lead to a "good" word, namely belong to the target language R. To put things in regular language terms, the intersection of the lan-

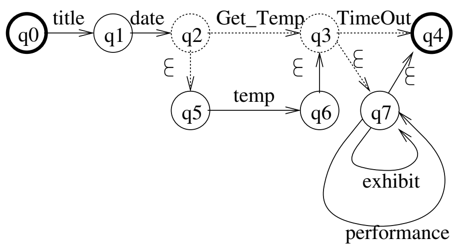

Figure 4: The A_w^1 automaton from the newspaper document.

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

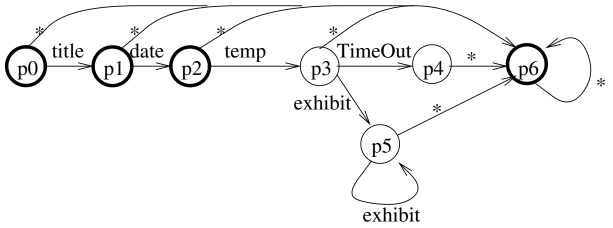

Figure 5: The complement automaton $\overline{A}$ for schema (**).

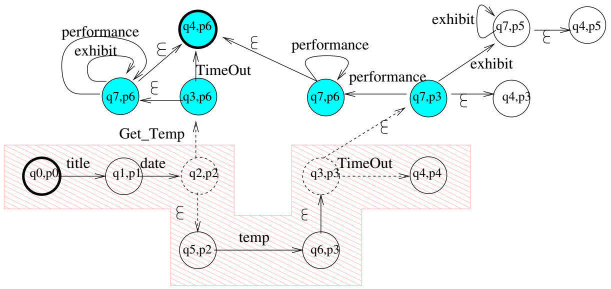

Figure 6: The cartesian product automaton $A_\times$.

guage 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 in (**) 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, resp.) is given in Figure 6. The initial state is $[q_0, p_0]$ and the final accepting one is $[q_4, p_6]$.

Note however that when searching for safe rewriting, one does not need to verify that all possible rewritings lead to a 'good word', i.e., that *none* 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 $\overline{A}$ is *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 non-marked. For the second 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 in (**). The safe rewriting sequence is the one obtained by following a non marked path. Each fork node on the path, together with its non-marked fork option, determines what needs to be done with the corresponding function - an ϵ edge

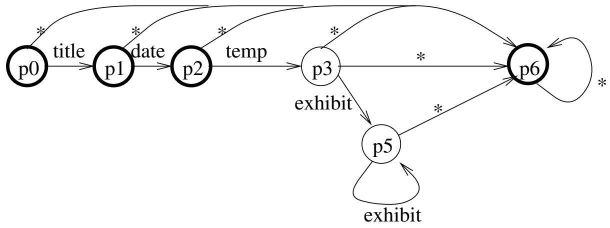

Figure 7: The complement automaton $\overline{A'}$ for schema (***).

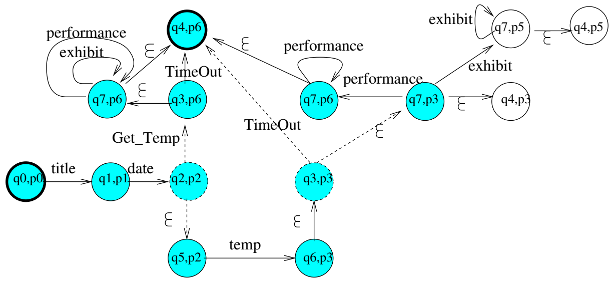

Figure 8: The cartesian product automaton $A'_\times$.

means "invoke the function" while a function edge means "do not invoke". In our example, it is easy to see (following the path with colored background) that Get_Temp needs to be invoked while $TimeOut$ should not.

For another example, consider the schema in (***). 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.

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 proposition states the correctness of our algorithm. We omit the proof for space reasons.

PROPOSITION 1. *The above algorithm finds a k-depth left-to-right safe rewriting if one exists.*

Complexity. We conclude this section by briefly discussing the complexity of the algorithm. Recall that we use s_0 to denote the schema of the sender and s to denote the agreed data exchange schema. The complexity of deciding whether a safe rewriting exists is determined by the size of the cartesian product automaton: we need to construct it and then traverse and mark its nodes. More precisely, the complexity is bounded by $O(\mid A_\times \mid^2) = O((\mid A_w^k \mid \times \mid \overline{A} \mid)^2)$. The size of A_w^k is at most $O((\mid s_0 \mid + \mid w \mid)^k)$ and the size of the complement automaton $\overline{A}$ is at most exponential in the automaton being complemented [11], namely at most exponential in the size of the target schema s. This exponential blow up may happen however only when s uses non deterministic regular expressions (i.e. regular expressions whose corresponding finite state

	possible rewriting (word w, functions output types $R_{f_1} \dots R_{f_n}$, target language R)
1	Build finite state automaton 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	**A rewriting *may* exist if the initial state is marked.**
7	To obtain such a rewriting:
8	(a) Follow a marked path 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).
9	(c) Backtrack when the calls return a value that does not allow to to continue to an accepting state.
10	(d) To minimize the rewriting cost, chose a path with minimal number/cost of function invocations.
11	exit

Figure 9: Possible rewriting of w into R

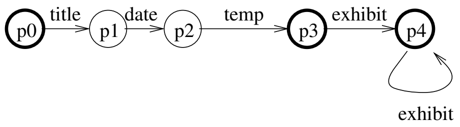

Figure 10: An automaton A for schema (***)

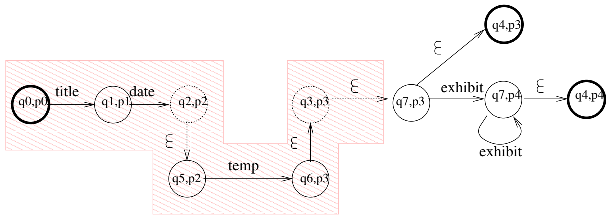

Figure 11: Cartesian product automaton for *possible* rewriting.

automaton is non deterministic). Note however that XML Schema enforces the usage of deterministic regular expressions only. 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).

The complexity of actually performing the rewriting depends on the 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 $w \times x^k$.

5. POSSIBLE REWRITING

We considered safe rewriting in the previous section. We now turn to possible rewriting. While function signatures provide an "upper bound" of the possible output, when invoked with the actual given parameters they may return a restricted "appropriate" output, so a rewriting that looked non feasible (unsafe) may turn to be possible after some function calls. To test if a rewriting *may* exist, we follow a similar three-steps 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 exactlyas before. For Step (3), Figure 9 provides an algorithm to test if the children of a given node *may* rewrite to the target schema. As before we use the automaton A_w^k that describes all the words that may be derived from the word w in a k-depth rewriting. w *may* rewrite to a word in the target language R iff some of these derived words belong to R. Namely, 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), that the final state is reachable from the initial one [11]. To find the actual rewriting, we follow an accepting path: We invoke functions, or not, as indicated by the *fork options* on the path, and backtrack when failing (i.e. when the function returns a value that does not correspond to the acceptance path.)

For instance, the automaton A for the schema in (***) with *newspaper* structure $title.date.temp.exhibit^*$ given in Figure 10. The initial state is p_0 and the final accepting states are p_3 and p_4. The cartesian product automaton $A_\times = A_w^1 \times A$ (for A_k^1 as in Figure 4 and A as in Figure 10) is given in Figure 11. The initial state is $[q_0, p_0]$. The final accepting states are $[q_4, p_3]$ and $[q_4, p_4]$, and all states (including the initial one) have an outgoing path to a final state. The only possible *fork options* left in the automaton, and which may lead to a possible rewriting, are the ones requiring the invocation of both Get_Temp and $TimeOut$ functions. If $TimeOut$ returns nothing but $exhibits$ the rewriting succeeds.

The correctness of this algorithm is stated below.

PROPOSITION 2. *The above algorithm finds a k-depth left-to-right rewriting, if one exists.*

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

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 mixed approach, that invokes some of the functions (e.g. ones with no side effects or low price) to get their actual output, while safely verifying other functions can be clearly beneficial. In terms of the algorithm in Figure 3 above, 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. The output of the invoked calls can also be later used in the actual rewriting (Details omitted)

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* the 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* the 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 that of documents. In fact, rather than testing *all* the schema instances (an infinite number) it suffices to look at a small number of *representative* documents. The key idea that that testing whether all elements of a given type have safe rewriting is analogous to testing whether a single function element, having output of that type, can be safely rewritten into the target schema. Thus, to check s_0, we need to look at one such function per element type in s_0, and test whether the document containing it as its single element can be safely rewritten. We omit the details for space reasons.

7. IMPLEMENTATION

The ideas and algorithms presented in the previous sections have been implemented and used in the *Schema Enforcement* module of the ActiveXML system [3, 1]. 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.

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 exploit the XML namespace [29] mechanism. More precisely, the namespace *http://www.vwx.com/namespace/int* is defined for function (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.vwx.com/namespace/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 </param>
  </int:params>
 </int:fun>
</newspaper>
```

Three attributes of the function nodes provide the necessary information to call the SOAP service: the URL of the server, the method name, and the associated namespace.

For defining the schemas for intensional documents, we use XML Schema$_{int}$, which is an extension of XML Schema. To describe intensional data, XML Schema$_{int}$ introduces *function* and *function patterns*. These are declared and used like elements and types in the standard XML Schema language. In particular, it is possible to declare functions and function patterns globally, and reference these declarations 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 some optional attributes and two optional sub-elements, *params* and *return*:

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

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

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

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

Functions are declared in a similar way as 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 $title.date.(Forecast \mid temp).(TimeOut \mid exhibit^*)$ may be defined in XML Schema$_{int}$ as:

```
<element name="newspaper">
  <complexType>
    <sequence>
      <element ref="title"/>
      <element ref="date"/>
      <choice>
        <functionPattern ref="Forecast"/>
        <element ref="temp"/>
      </choice>
      <choice>
        <servicePattern ref="TimeOut"/>
        <element ref="exhibit" minOccurs="0"
                 maxOccurs="unbounded"/>
      </choice>
  </complexType>
</element>
```

Similarly to XML Schema, we require the definitions to be unambiguous [29]. Namely, when parsing a document, for each element and each function node, the sub-elements can be sequentially assigned a corresponding type/function pattern in a deterministic way by looking only at the element/function name.

We extend WSDL in the obvious way. One of the major features of the WSDL language is to describe the input and output types of Web services function using XML Schema. We simply extend it by 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 (by using attribute `WSDLSignature`).

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 on top of the repository documents. All the exchanges between the ActiveXML peers, and also with other Web service providers/consumers use the SOAP protocol.

The important point here is that both the services that a ActiveXML peer invokes and those that it provides can 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$_{int}$ description. Similarly, when calling a ActiveXML peer, the parameters of the call should comply with its interface. 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, raise an exception and report an error. Similarly, before a ActiveXML service returns its answer, the module performs the same three steps on the returned data.

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 develop an implementation from scratch [22]. Whereas the first solution seems preferable, we opted for the second one because, at the time we started the implementation, the available (free) software we tried (Apache Xerces [28] and Oracle Schema Processor [23]) showed to be extensible only in a limited way. Our parser uses a standard SAX parser [28]. 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 all simple types, inheritance and keys. These 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 a very efficient variant of the algorithm depicted in Figure 3. The algorithm in the figure starts by constructing all the required automaton and only then analyzes the resulting graph. By contrast, our implementation builds the automaton in a lazy mode, starting from the initial state, constructing only the needed parts. The construction is pruned whenever a node can be marked directly, without looking at the remaining, unexplored, branches. The two main ideas that guide this process are the following:

Sink nodes Some accepting states in $\overline{A}$ are "sink" nodes - once you get there you cannot get out (e.g. p_6 in Figures 5 and 7). When such a node is reached in the construction of the Cartesian product automaton $A_\times$, we can immediately mark the node and prune the outgoing branches. E.g., in Figure 12, 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 declared 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 further its outgoing branches. To continue with the above example, once 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 at [22].

8. 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

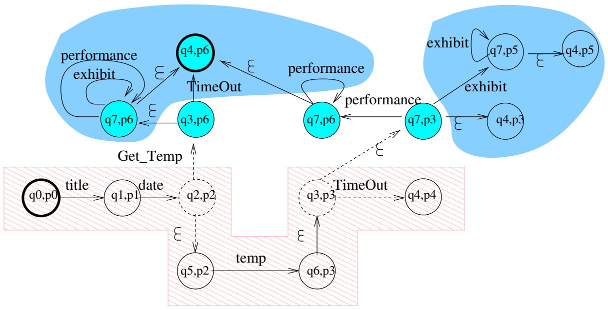

Figure 12: The pruned automaton.

not a new idea. Functions embedded in data were already present in relational systems [21] as stored procedures. Also, method calls form a key component of object-oriented databases [5]. In the Web context, scripting languages such as php [24] or JSP [15] 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 documents [23]. The embedding of Web service calls in XML documents is also part of 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 towards 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, e.g., [17, 12, 6]. We selected XML Schema [29] for several reasons. First, XML Schema is the current standard recommended by the W3C for describing the structure of XML documents. Furthermore, it is the typing used in the WSDL language for defining SOAP service signatures [27]. 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 [4]. More specifically for us, the embedding of Web service calls into 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. Security and access models for XML and Web services have been proposed in, e.g., [7, 18]. The control of the calls to Web services in intensional XML Schemas is somewhat complementary as illustrated by the example in Section 2. Based on our still limited experience, we can say that it is quite useful in real applications. However, more work is needed to better integrate this aspect with other aspects of security.

The work presented here is part of the ActiveXML [3, 1, 2] project based on XML and Web services. We presented in this paper 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, e.g., UDDI style search [26], to try to find services on the Web providing 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 [2]. Note that when data may be found in different places and a service may be performed in 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 [25] and to distributed computing at large. The novel aspect is the ability to exchange intensional information. This is in spirit of [14], that consider also the exchange of intensional information in a distributed query processing setting.

Intensional XML documents fit nicely in the context of data integration since an intensional part of an XML documents may be seen as a view of some data source. The Web services in XML data may be used for wrapping Web sources [9] or for propagating changes for warehouse maintenance [31]. Note that the control of whether or not to materialize data (studied here) provides some flexible form of integration that is a hybrid of the warehouse model (all is materialized) and the mediator model (nothing is). On the other hand, this is orthogonal to the issue of selecting the views to materialize in a warehouse, studied in, e.g., [10, 30].

To conclude, we mention some fundamental aspects of the problem we studied. Although the k-bound 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. First, one could try to find, for the safe rewriting problem with the k-bound restriction, algorithms with a better worst-case complexity and derive a tight lower bound for the problem. Also, the main open problem is whether safe rewriting remains decidable when the k-depth restriction is removed. One could try also to improve the results for schema validation, possible or mixed rewriting.

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 [19] that are based on tree automata. We are also considering the use of tree automata to attack the main open problem.

9. REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active xml: Peer-to-peer data and web services integration (demo). VLDB, 2002.

[2] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, and Tova Milo. Dynamic xml ducuments with distribution and replication. SIGMOD, 2003.

[3] Active xml. `http://www-rocq.inria.fr/verso/Gemo/Projects/axml/`.

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

[5] R.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufman, 1996.

[6] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proc. of ACM SIGMOD*, pages 177–188, 1998.

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

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

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

[10] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Proc. of ICDT*, pages 98–112, 1997.

[11] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

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

[13] Jelly: Executable xml. `http://jakarta.apache.org/commons/sandbox/jelly`.

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

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

[16] Macromedia Coldfusion MX. `http://www.macromedia.com/`.

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

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

[19] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. In *Proc. of ACM PODS*, pages 11–22, 2000.

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

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

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

[23] Oracle XML Developer's Kit for Java. `http://otn.oracle.com/tech/xml/`.

[24] The PHP Hypertext Preprocessor. `http://www.php.net`.

[25] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition*. Prentice-Hall, 1999.

[26] Universal Description, Discovery, and Integration of Business for the Web (UDDI). `http://www.uddi.org`.

[27] Web Services. `http://www.w3.org/2002/ws`.

[28] The Xerces Java Parser. `http://xml.apache.org/xerces-j/`.

[29] XML. `http://www.w3.org/XML`.

[30] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *The VLDB Journal*, pages 136–145, 1997.

[31] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proc. of ACM SIGMOD*, pages 316–327, 1995.