# Active XML: A Data-Centric Perspective on Web Services


**Serge Abiteboul**
**INRIA**
**and Xyleme**

**Omar Benjelloun**
**INRIA**

**Ioana Manolescu**
**INRIA**

**Tova Milo**
**INRIA**
**and Tel Aviv U.**

**Roger Weber**
**ETH Zurich**



Paper ID: 379



Contact author: Omar Benjelloun, INRIA Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France
Tel: (+33)1 39 63 52 83; fax (+33) 1 39 63 56 74
E-mail: Omar.Benjelloun@inria.fr



**Program Committee Member: Tova Milo**

## Area: Infrastructure for Information Systems

## Category: Research

**Topics:**
Databases and database services in new context - Internet and the WWW,
Data Transformation, Integration, Evolution, and Migration,
Interoperability, Heterogeneous and Federated Databases, Mediators

# Active XML: A Data-Centric Perspective on Web Services

**Serge Abiteboul** INRIA and Xyleme
**Omar Benjelloun** INRIA
**Ioana Manolescu** INRIA
**Tova Milo** INRIA and Tel Aviv U.
**Roger Weber** ETH Zurich

## Abstract

We propose a peer-based architecture that allows for the integration of distributed data and web services. It relies on a language, Active XML, where (1) documents embed calls to web services that are used to enrich them, and (2) new web services may be defined by XQuery queries on such active documents.

Embedding calls to functions or even to web services inside data is not a new idea. Our contribution, however, is turning them into a powerful tool for data and services integration. In particular, the language includes linguistic features to control the timing of service call activations. Various scenarios are captured, such as mediation, data warehousing, and distributed computation. A first prototype is described.

## 1 Introduction

One of the most essential issues in computer science is the management of data. Since the 60's, the database community has developed the necessary science and technology to manage data in central repositories. From the early days, many efforts have been devoted to extending these techniques to the management of distributed data as well, and in particular to its integration, e.g., [23, 43, 37]. But the web is dramatically changing the context for data integration, mainly because of (i) the high heterogeneity and interoperability problems between sources and (ii) the independence of sources and the very large scale of the web. We propose in this paper a new framework for data and service integration, that addresses these two concerns.

**Heterogeneity and interoperability issues** XML, as a self-describing semi-structured data model, raised large interest among the data integration community, e.g. in [19, 29, 33] for the flexibility and expressiveness it brings for solving semantic heterogeneity issues. However, a significant interoperability problem comes from the fact that data is often dynamically constructed by programs, e.g., upon the submission of a form by a human user. This "deep web" is made accessible by *Web services* which encapsulate programs with XML arguments and results. The SOAP [36] and WSDL [44] languages enable calling and describing these remote programs seamlessly. *In our framework, web services are effectively leveraged for data integration tasks.*

**Independence and scalability issues** Centralized architectures for data integration somehow contradict the essence of the web, promoting distribution and independence. Furthermore, they have difficulties scaling up to its large size. Peer-based architectures, in which resources are shared by direct exchange between systems, propose a credible alternative and are already spreading, mainly in a file-sharing context [31, 21, 20]. *In our framework, peer-based architectures form the basis for scalable data integration.*

We propose Active XML (AXML in short), a language that leverages web services for data integration and is put to work in a peer-to-peer architecture. The language enables embedding service calls inside an XML document, that is enriched by their results when they are fired. The language also allows for declarative specification of new web services based on such active documents and a query language. Data with embedded calls to operations is an old idea that has already been considered in the context of XML and web services. For instance, in Microsoft Office XP, Smart Tags within XML documents can be linked to Microsoft's *.NET* platform for web services [35]. However, to our knowledge, the present paper is the first proposal that actually turns calls to web services embedded in XML documents into a powerful tool for data integration. By providing means to declaratively specify when the service calls should be activated (e.g. when needed, every hour, etc.), and for how long the results should be considered valid, our approach allows to capture and combine different styles of data integration, such as warehousing, mediation and flexible combinations of both. Moreover, AXML allows to use and specify novel kinds of web services:

**Continuous services** Most existing services are in the style of remote procedure calls (RPC): they are

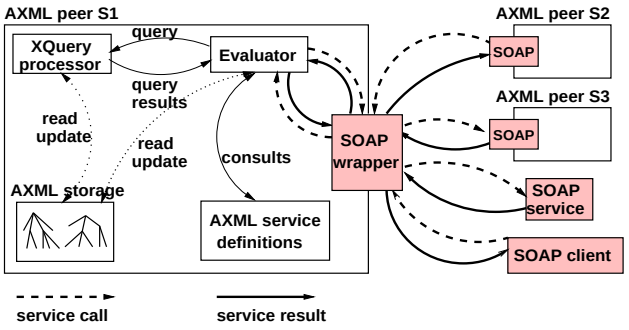


Figure 1: Outline of the AXML data and service integration architecture.

called with some parameters and (eventually) return an answer. By contrast, *continuous* services return a (possibly infinite) stream of answers for a single call. As an example, consider subscription systems where new data of interest are pushed to users (see, e.g., [32]). Similarly, a data warehouse receives streams of updates from data sources, for maintenance purposes. Streams of results are also generated, for instance, by sensors (e.g., thermometers, video surveillance cameras). The AXML framework allows to support the management (production and integration) of data for such continuous services.

**Services with intensional input/output** Standard web services typically exchange plain XML data. We allow web services to exchange AXML data that may contain calls to other services. In the spirit of [27], the answer to an AXML service call may include both extensional and intensional information. Furthermore, the arguments of a call may also contain service calls (intensional information). This allows, for instance, to delegate portions of a computation to other sites, i.e., to distribute computations.

As mentioned above, AXML allows not only to use existing web services but also to define new ones on top of AXML documents. The service specifications are based on XQuery [48], the future standard query language for XML, extended to updates, and allow in particular the definition of continuous services and of services with intensional input/output.

AXML is put to work in a peer-based architecture, illustrated by Figure 1. Each *peer* contains a repository of AXML documents as well as some AXML web services definitions. At the heart of the system is an AXML processor and, in particular, an Evaluator module which is in charge of the following two main tasks:

**Client** Choosing which of the service calls embedded in the AXML documents need to be fired at each point in time, firing the calls, and integrating the returned answers into the documents. It is important to stress that any web service can be used, be it provided by an AXML peer or not, as long as it has a SOAP interface[1].

**Server** Accepting requests for the AXML services supported by the peer, executing the services (i.e. evaluating the corresponding XQuery) and returning the result.

Observe that since AXML services query AXML documents and can accept (resp. return) AXML documents as parameters (resp. result), a service execution may require the activation of other services calls. Thus, these two tasks are closely inter-connected.

The paper is organized as follows: After an overview of related work, the AXML language is presented in Section 3, mainly through an extended example. A formal semantics for AXML documents and services is presented in Section 4, while security and peer capabilities are considered in Section 5. An evaluation strategy and an implementation are discussed in Section 6. The last section is a conclusion.

## 2 Related work

The starting point of the present work is the semistructured data model and its current standard incarnation, namely XML [45]. We rely primarily on an XML query language and on protocols for enabling the remote procedure calls on the web. Disparate efforts to define a query language for XML are now unifying in the XQuery proposal [48]. The core of our definition of web services will consist of parameterized queries in XQuery. The various industrial proposals for web services architectures, e.g. .NET by Microsoft, e-speak by HP, SunOne by Sun Microsystems, are also converging towards using a small set of specifications to achieve interoperability. Among those, we are directly concerned by the Simple Object Access Protocol (SOAP) [36] and the Web Services Description Language (WSDL) [44], that are used in our implementation. More indirectly, UDDI registries [39], can be leveraged by our system, e.g., to publish or discover web services of interest.

As already mentioned, the idea of embedding function calls in data is not a new idea. Embedded functions are already present in relational systems [40], e.g., as stored procedures. Method calls form a key component of object databases [12]. The introduction of service calls in XML documents is thus very natural. Indeed, external functions were present in Lore [26] and an extension of object databases to handle semistructured data is proposed in [33], thereby allowing to introduce external calls in XML data. Our work is tailored to XML and web services. In that sense, it is more directly related to Microsoft Smart Tags [35]. Our goal is to provide means of controlling and enriching the use of web service calls, and to equip them with a formal semantics.

The activation of service calls is closely related to the use of triggers [40] in relational databases, or rules in active databases [42]. Active rules were recently

[1] Bridges to alternative protocols for web services, such as XML-RPC, may clearly be used.

adapted to an XML and XQuery context [8]. A recent work considered firing web service calls [9]. We harness these concepts to obtain a powerful data integration framework based on web services. In some sense, the present work is a continuation of previous works on *ActiveViews* [1]. There, declarative specifications allowed for automatic generation of applications where users could cooperate via data sharing and change control. The main differences with ActiveViews are that (i) AXML promotes peer-to-peer relationships vs. interactions via a central repository, and (ii) the cornerstones of the AXML language are XPath, XQuery and web services vs. object databases [12].

Integration and composition of web services have been active fields of research recently [41]. However, most of the works have focused on process-oriented techniques, and on the definition of workflows among services [13]. In Multilisp [24], intensional data was used under the form of "futures", i.e., handles to results of not-yet-finished computations, to allow for parallelism. Ambients [10, 11], as *bounded* spaces where computation happens, also provide a powerful abstraction for processes and devices mobility on the web. The present work is concerned with the complementary data-centric aspects of web service integration.

Our formal foundation is based on nondeterministic fixpoint semantics [3], that was primarily developed for Datalog extensions. In that direction, the paper has also been influenced by recent works on distributed Datalog evaluation [27].

## 3 AXML by example

In this section, we introduce *Active XML* via an example. In Section 3.1, we present a simple syntax for including service calls within AXML documents, and outline its core features. Section 3.2 deals with intensional parameters and results of service calls. We then consider, in turn, the lifespan of data, the activation of calls and the definition of AXML services.

### 3.1 Data and simple service integration

At the syntactic level, an AXML document is an XML document. At the semantic level, we view an AXML document as an *unordered data tree* i.e., the relative order of the children of a node is immaterial. While order is important in document-oriented application, in a database context like ours it is less significant and we assume that, if needed, it may be encoded in the data. Also, we attach a special interpretation to particular elements in the AXML document that carry the special tag <sc>, for *service call*; these elements represent service calls that are embedded in the document[2]. In general, a peer may provide several web services. Each service may support an array of functions. We use here the terminology *service call* for a call to one of the functions of a service.

As an illustration, consider the AXML document corresponding to my personal page for auctions that I manage on my peer, say mypeer.com. This simple page contains information about categories of auctions I am currently interested in, and the current outstanding auction offers for these categories. The page may be written as follows:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc>auction.com/getOffers("Toys")</sc>
 </category>
 <category name="Glassware">
   <sc>eBay.com/getAuctions("Glassware")</sc>
 </category>
</axml>
```

While the category names are explicitly written in the document, the offers are specified only *intensionally*, i.e., using service calls instead of actual data. Here, the list of toy auctions is provided by auction.com. On that server, the function getOffers, when given as input the category name Toys, returns the relevant list of offers, as an XML document. The latter is merged in the document, which may now look as follows:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc>auction.com/getOffers("Toys")</sc>
   <auction aID="1">
     <description>Stuffed bear toy</description>
   </auction>
   <auction aID="2">...
 </category>...
</axml>
```

Observe that the new data is inserted as sibling elements of <sc>, and that the latter is *not* erased from the document, since we may want to re-activate this call later to obtain new auction offers. Finally, note that in the case of a *continuous* service, several result messages may be sent by the service for one call activation. In this case, all the results accumulate as siblings of the <sc> element.

**Merging service results** More refined data integration may be achieved using ID-based data fusion, in the style of e.g., [34, 4, 17]. In XML, a DTD or XML Schema may specify that certain attributes uniquely identify their elements. When a service result contains elements with such identifiers, they are merged with the document elements that have the same identifiers, if such exist. To illustrate this, assume that auction.com supports a getBargains function that returns the list of the current ten most attractive offers, each with its special bargain price. Suppose also that aID is a key for auction elements. If an auction element with aID "1" is returned by a call to getBargains, the element will be "merged" with the auction with aID "1" that we already have.

[2]For readability, we use a simple syntax for <sc> elements. A more complete version is provided in appendix.

**XPath service parameters** The parameters of a service call may be defined extensionally (as in the previous examples) or may use XPath queries. For instance, the getOffers service used above gets as input a category name. Rather than giving the name explicitly, we can specify it intensionally using a relative XPath expression [47]:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc>auction.com/getOffers([../@name/text()])</sc>
   ...</category>...
</axml>
```

The XPath expression ../@name/text() navigates from the <sc> node to the parent <category> element, and then to its name attribute. In this example there is only one possible instantiation for the XPath expression. In general, several document subtrees may match a given XPath expression. When this is the case, *the call is activated once for each possible instantiation* (if a call has several XPath parameters, we have one activation for every tuple in the cross-product of the paths instantiations).

Besides the parameters, the name of the called peer as well as the name of the service itself may be specified using relative XPath expressions. The same cross-product semantics applies.

### 3.2 Intensional parameters and results

The parameters of the services calls that we have seen so far were (instantiated as) simple strings. In general, the parameters of a service call may be arbitrary AXML data, specified either explicitly, or by an XPath expression. In particular, AXML parameters may contain calls to other services, leading thus to *intensional* service parameters. For example, to get a more adequate set of auctions, we may use a service that, in addition to the category name, needs the current user budget, which is itself obtained by a call to the bank services:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc>auction.com/getOffers1(
     [../@name/text()],
     <sc>bank.com/getBudget("user25")</sc>)
   </sc></category>
</axml>
```

Up to now, we have not discussed where and when a service call is activated. In the above example, we already face a choice concerning the activation of getBudget. We may call it first, and then call getOffers providing it with the result. Another option is to call directly getOffers with the intensional parameter, and let it handle the activation of the call to getBudget. We will further discuss this issue in Section 5.

Services may not only get intensional data (i.e. AXML documents with embedded service calls) as input, but also return such data as a result. As an example, each auction in the result of getOffers may contain a call to a getDetails service that provides more information about that particular auction:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc>auction.com/getOffers([../@name/text()])</sc>
   <auction aID="1>
     <description>Stuffed bear toy</description>
     <sc>auction.com/getDetails([../@aID])</sc>
   </auction>...</category>...
</axml>
```

Observe that intensional results already appear in practice in many popular applications. For example, the Google search engine returns, for a given keyword, some document URLs plus (possibly) a handle for obtaining more answers. With this handle, one can obtain a new list and perhaps another handle.

### 3.3 Controlling the lifespan of data

So far, all the service call results were accumulated in the calling document. In practice, we need more flexibility in managing these results, so that we may replace old results with new ones, discard data that has grown too old or has become inconsistent, etc. Many models and techniques have been proposed for managing data lifespan, particularly in the fields of version management, temporal databases, and active databases. For our purposes, we choose a suitable, simple model, that may be extended with more complex features.

To manage data lifespan, we conceptually attach a special attribute expiresOn to any data node in an AXML document[3]. Some nodes may have explicit expiration time, whereas others will inherit it from their parent. Expired nodes should simply be viewed as erased from the document.

The value of the expiresOn attribute is an event, that may depend on time, and/or on the document content. For example, if a user wants to specify that her interest in a product category lasts only until February 19th, 2002, then the element will have the following form:

```
<category name="Toys" expiresOn="Feb. 19th, 2002">...
```

Data returned by a service may come with a expiration time specification. This is a very useful feature that allows a service provider to state how long the particular result is meaningful. For example, getOffers may inform the user of an auction's closing time, by setting expiresOn for the returned data. The lifespan of a service call result may be explicitly *overwritten* by the caller. This is done using a valid attribute, in the sc element. valid can be a function of the time when the call was (last) answered, denoted $rt$. For example, consider the following two calls:

```
<sc valid="rt + 1 year">
   auction.com/contGetOffers("A")</sc>
<sc valid="last 5">
   auction.com/contGetOffers("B")</sc>
```

[3]Strictly speaking, it is not possible in XML to attach an attribute to a #PCData node. It is possible to do so in AXML.

The first one states that auctions in Category $A$ are archived for one year, whereas the second one requires to keep only the last 5 of Category $B$.

### 3.4 Controlling the activation of calls

To control *when* a service call is activated, we use two attributes of <sc> elements, namely *mode* and *frequency*. The value of the frequency attribute is similar to the one of valid, except that it is a function of *ct*, the time when the service was last called. Thus, we can easily specify a given instant, a time interval, the occurrence of an event, etc. By default, a service is called only once, when the document is registered. We say that a call has *expired* when, according to its frequency attribute, it should be activated.

The mode of a call is either *lazy* or *immediate*. In immediate mode, the call is activated when it expires; if the call is in lazy mode, the fact that it has expired only means that next time the data produced by this call is needed (e.g., by a query over the AXML document), the service has to be called.

The data validity and the calls activation mode and frequency, together, provide a flexible and powerful tool for capturing various integration scenarios. This is illustrated next. In the following integration styles, the first three assume regular (non continuous) services whereas the last one relies on continuous services:

- mediator style: set valid to 0 (note that an immediate mode would not have any meaning in this case).
- mediator style with caching: choose a non-zero value for valid, and lazy mode.
- warehousing mode with pulling information: choose valid larger than frequency, and immediate mode.
- warehousing mode with pushing: choose a non-zero value for valid, and immediate mode.

Note that the activation of a call is dissociated from the lifespan of its results. For example, if we wish to call getOffers every day, and keep the results for a week after their acquisition, we would write:

```
<sc valid="rt + 1 week" frequency="ct + 1 day">
    auction.com/getOffers("Toys")</sc>
```

Also, some form of semantic consistency may be enforced by linking call activation to change events. A user may want the set of auctions on her page to follow changes in her interests. In other words, when the name of the category of interest is updated, the offers returned by the previous call become irrelevant and the getOffers service needs to be re-activated with the new category name to obtain the relevant offers. This can be written as follows:

```
<axml id="user25"> Welcome to mypeer.com !
 <category name="Toys">
   <sc valid="until param changes">
       frequency="when param changes">
       auction.com/getOffers([../@name/text()])
   </sc></category>
</axml>
```

The keyword param links the data validity/call activation to changes in the value of the call parameters. In general one can use there an arbitrary XPath expression to link the validity/activation to changes in other elements of the document.

**Remark 3.1** (timeout) In the case of a non-continuous service, it may happen that the answer returns very late, or never returns at all. In practice, it is useful to have *timeouts* for calls. When the timeout is reached, the system abandons hope of getting the result. A real system should also provide an exception handling mechanism to manage such events. □

### 3.5 AXML service definition

The AXML framework allows to call arbitrary web services. It may also be used to define new web services, as illustrated in this section. In short, an AXML service is defined by a parameterized XQuery queries over the peer's AXML documents. As an example,getOffers, that returns all currently open auctions for a given category, may be defined at auction.com as follows:

```
let sc auction.com/getOffers($c) be
 for $cat in document("auction.com/a.xml")//category
     $a in $cat/auction,
     $aID in $a/@aID/text(),
     $des in $a/description/text()
 where $cat/@name/text()=$c
 return <auction aID={$aID}>
          <description>{$des})</description>
          <sc>auction.com/getDetails({../@aID})</sc>
        </auction>
```

In the above example, the category parameter $c is of type #PCDATA (text). The query consults an AXML document (a.axml) which may contain service calls, and constructs an AXML document with some calls (e.g. to the getDetails function of auction.com). Here again we face a choice concerning the activation of getDetails: we may call it first, and only then return the answer of getOffers. Another option is to return the document immediately and let the caller of getOffers handle the activation of the calls to getDetails. The particular type of the service result may be described by an XML Schema [46], as described by the WSDL specification [44].

To define *continuous* AXML services, we simply prefix the definition with the keyword continuous. Thus, a continuous variant of a getOffers, returning the set of interesting auctions whenever it changes, is defined as follows:

```
let continuous sc auction.com/contGetOffers($c) be...
```

To define AXML services with side effects, in the absence of a standardized language for XML updates, we use the extension to XQuery proposed in [38]. For example, the "a.xml" database that getOffers consults may be enriched by the addAuction service:

```
let update sc auction.com/addAuction($c, $id, $d, $p) be
 for $cat in document("auctions.com/a.xml")//category
 where $cat/@name/text()=$c
 update $cat { insert <auction aID={$id}>
                        <description>{$d}</description>
                        <startPrice>{$p}</startPrice>
                      </auction> }
```

### 3.6 Discussion

We conclude this section with two remarks regarding consistency and termination.

**Consistency** We assume that the document we start with is well-formed and obeys its DTD (or XML Schema) if one is specified for it. An inconsistency may arise if one call leads to constructing a document that no longer obeys the schema. While some of this may be prevented by consulting the declared signature of the used services [25, 6], static type checking becomes more complicated due to the use of ID-based element fusion and of XPath expressions in call parameters.

**Termination** We have seen above that a service calls may return intensional answers. Note that this may lead to a non terminating computation: the result of a service call may contain new service calls that need to be activated. Those in turn may return new calls to be activated, and so on. Another possible reason for non termination is mutual recursion between service calls. Some simple sufficient conditions for termination are mentioned in the Appendix.

## 4 Data and computation model

In this section, we briefly define the AXML data model and the semantics of AXML documents and services. For lack of space, the presentation is informal. The formal definitions as well as the proofs of the results can be found in [2].

Intuitively, an AXML instance consists of a number of peers, each one containing some AXML documents that are being run. AXML documents are XML unordered trees. The evaluation of these documents generates calls between these peers and possibly results in new documents being evaluated at each peer. As we shall see, the evaluation is non-deterministic, which captures the asynchronous evolution of the global instance, which may eventually reach a fixpoint or not. More precisely, we present next the data model, then the computation.

### 4.1 Data model

An (AXML) instance consists of a number of peers. Each peer contains AXML documents, some service definitions, and a working area. We next define instances, then proceed to the definition of documents and services.

**Instances** An *instance* $\mathcal{I}$ consists of a number of peers $p_1, \ldots, p_n$. The *content* of a peer $p_i \in \mathcal{I}$ is defined by a triple $(\mathcal{D}_i, \mathcal{F}_i, \mathcal{W}_i)$ where $\mathcal{D}_i$, the peer *documents*, is a set of AXML documents, $\mathcal{F}_i$, the peer *services*, is a set of AXML service definitions, and $\mathcal{W}_i$, the peer *working area*, is a set of AXML documents. All the sets are assumed to be finite.

The documents in $\mathcal{D}_i$ form the persistent repository of the peer. Each document $d$ in the working area of a peer $p_i$ represents the computation of some service call in $p_i$, i.e., some current work that $p_i$ is performing. Any such document $d$ also contains a *destination* attribute specifying where the result of this computation should be sent, including the identity of the calling peer and the parent of the service call element.

**Documents** As for standard XML documents, an AXML document is modeled by a labeled tree with nodes representing the document elements/attributes and with edges capturing the component-of relationship among document items. The three main differences with the standard XML data model [45] are that (1) we ignore here the order of elements, hence our trees are unordered[4], so we only consider the order-free fragments of XPath (for parameters) and XQuery (for service definitions); (2) a validity predicate is attached to some elements to specify when some particular data become stale; (3) some of the tree leaves are special service call nodes, called in the sequel *sc-nodes*. An sc-node is labeled by a tuple of the form $\langle s, f, p_1, \ldots, p_n \rangle$ where:

- $f$ and $s$ are respectively a *service* and *peer* names, or are XPath expressions. In the first case, the service $f$ must be defined in peer $s$ with arity $n$.
- $p_1, \ldots, p_n$, the call parameters, are AXML documents or XPath expressions.

An sc-node where none of $s, f, p_1, \ldots, p_n$ are XPath expressions, is called a *concrete* call.

**Reduced documents** Continuous services send a sequence of answers to the caller. Smart (or optimized) such services may only send the delta since the last answer. In other cases, the caller may be responsible for detecting and ignoring redundant data. To abstract this (without having to get into implementation details such as who performs the optimization and when), we use in the formal model the notion of *reduced* version of a document, where multiple occurrences of the same data are omitted.

To define reduced documents, we use the auxiliary concept of *inclusion relationship* among trees. A reduced document is such that no subtree is included in one of its siblings. One can show that the reduced version of a document is unique. It can be, for instance, computed by iteratively removing redundant subtrees. The details are omitted. We will assume in the sequel that all our AXML documents are reduced.

[4] We may take into account the ordering in some specific cases, e.g., for the extensional portions of documents.

**Service definitions** To conclude this section, let us consider the definition of $\mathcal{F}_i$, i.e., the definition of services. The semantics of XQuery queries is standard, with one notable exception: when evaluating path expressions, service calls act like document boundaries which the evaluation cannot cross. In other words, they are terminal nodes which do not match any path expression.

The definition of an AXML service consists of the service name, the service type (e.g. continuous or not), the service parameter names $v_1, \ldots, v_n$, and a parameterized query $Q(v_1, \ldots, v_n)$, namely a query that may refer to the (parameters) documents $v_1, \ldots, v_n$.

### 4.2 Computation

We are now ready to define the semantics of AXML documents. Each peer includes a collection of AXML trees, in $\mathcal{D}_i$ and $\mathcal{W}_i$. These documents may contain service calls that may be activated to derive more information about the documents. A service call activation spans a computation on one of the peers. More precisely, the activation in Peer $s$ of a particular service call to Peer $r$ involves (1) (possibly) instantiating in $s$ the XPath expressions of attributes of the call, (2) for each instantiation, sending concrete calls from Peer $s$ to Peer $r$, (3) computing in Peer $r$ the corresponding answers and (4) returning the answers to Peer $s$, where they are received and merged at the appropriate place in the tree. If, for some reason, the resulting tree is no longer a legal AXML document, it becomes the *inconsistent* document.

Recall that the decision whether service calls can, and need to, be instantiated (resp. sent, computed, returned) at a given time depends on the specific call attributes. We will simply refer to such calls as *eligible for instantiation* (resp. *sending*, *computation*, *returning*). (We will see in the next section how this can be implemented.)

An *initial instance* is such that all peers have an empty working area. Given an initial instance $\mathcal{I}$, each peer $s = \langle \mathcal{D}, \mathcal{F}, \mathcal{W} \rangle$ evolves in a similar way. Starting from $\mathcal{I}$, repeatedly (and non-deterministically), one of the following steps is executed:

**Step 1: Instantiate the XPath parameters:** For some (non concrete) sc-node $v$ in $\mathcal{D}$ or $\mathcal{W}$ that is eligible for instantiation, the XPaths are evaluated, and for each instantiation, a new document is added to $s$'s Working Area. The roots of these documents have the corresponding concrete service call as an sc-node child, and have $v$ as the destination for the result of the computation.

**Step 2: Send/Receive an external call:** For some concrete sc-node $n$ in $\mathcal{D}$ or $\mathcal{W}$ that is labeled with a call $c$ to some remote peer $r$ and is eligible for sending, the call is activated. Formally, this consists in adding, to the Working Area of the receiving peer $r$, a new document whose root has an *sc-node* child labeled with $c$ and having $n$ as the destination for the result.

**Step 3: Compute a local call:** For some concrete sc-node $n$ in $\mathcal{D}$ or $\mathcal{W}$ that is labeled with a call $c$ to a local service of $s$ and is eligible for computation, evaluate the service query using the given parameters. The result, a forest, is merged under the parent node of $n$.

**Step 4: Return/Receive result of a call:** For some document $d$ in $W$, eligible for being returned as an answer, the children of $root(d)$, (not including $d$'s *destination* attribute) are sent to the destination peer and merged under the parent of the destination node.

Observe that, in the above computation, we grouped *sending* (resp. *returning*) a call and *receiving* it in one operation. Intuitively, our send/receive (resp. return/receive) operation captures the moment when the receiver receives the message. Finally, to guarantee a correct semantics, we need some *fairness* condition:

> (†) Any operation that may happen, eventually happens.

**Non-determinism and confluence** In general, an initial instance $\mathcal{I}$ may be transformed in many different ways, depending on the choice of the operations to perform. This non-determinism is built in the semantics. So, even if an instance converges to a fixpoint, the fixpoint does not have to be unique. Furthermore, as mentioned in Section 3.6, the computation may continue forever, building more and more data, i.e., there is no guarantee of termination.

Although this may seem to be a negative feature of the model, observe that this naturally models the real world we are trying to capture. The state of a peer may continuously evolve because, for instance, of interactions with human users updating data. Also, continuous external services such as subscriptions may keep sending new data to the peer. So, the system should not be expected to terminate. Also, data may expire or get deleted and the order in which the various operations/queries are executed may have impacts on the state. Thus, because of asynchronicity and peer independence, determinism is an elusive goal in such an environment. However, termination and confluence can be enforced under very strict restrictions, as outlined next.

**Remark 4.1** (Monotone computation) Suppose the computation is monotone, i.e., no fact is ever deleted or updated, and information keeps being added in a cumulative manner. Under this restrictions, the order in which the steps are executed is not significant anymore. One can then guarantee that all computations lead to a (possibly infinite) unique state. This is in the spirit of results on inflationary fixpoint semantics, see [3]. Details are deferred to the appendix. □

## 5 Limiting the firing of calls

So far, we have described the AXML paradigm at a rather abstract level. Before we consider its actual implementation, we highlight some important issues that are critical for a real life implementation. Before activating a service call, two points need to be checked: (i) that the receiving peer is willing to accept the call, i.e., that the caller has the proper priviledges to issue the call, and (ii) that the peer has the capability to process the call, i.e., that the parameters of the call can be understood by the receiver. In practice, access to services from other peers will be severely controled for security reasons. Also, peers will have limited capabilities, e.g., most of them will only accept calls with "plain" XML arguments. This is the topic of this section.

### 5.1 Site capabilities and security

First, consider peer capabilities. We illustrated in Section 3.1 the use of intensional parameters in a service call. Observe that they may, in principle, be evaluated *before* or *after* the call is sent to the receiving peer. In practice, not all choices are always feasible. For instance, consider again the example in Section 3.1. If auction.com is not capable of calling getBudget on bank.com (e.g., because it is not an AXML peer), then "user25"'s AXML peer must first call getBudget, and only then call getOffers with the result.

Now, consider the security concerns that must guide call activations. Access control is a needed features for many applications. First, a service provider may wish to reserve the access to a service to those who paid for it. For example, acm.org currently allows users from the inria.fr domain to use the search services of the ACM digital library, but not any web user can do so. Furthermore, security is necessary to protect sites from a malicious usage. Not surprisingly, the exchange of data that includes service calls is a major security hole. For instance, suppose that we want to break into a peer $s$, say the site *qod.com*, providing quotes of the day. There are two main ways to do this:

**In a call parameter** Intensional service parameters open backdoors to AXML servers. For instance, a malicious client may use the following call to *qod*:

```
<sc>qod.com/QuoteOfDay(
    <sc>buy.com/BuyCar("BMW Z3")</sc>)</sc>
```

This malicious user does not wish to buy the car by himself, but tries to make qod.com buy it.

**In a call result (Trojan Horse)** Suppose now that *qod.com* is malicious in the quotes it provides, e.g., by returning the following quote as a call result:

```
<quote> Love means never having to say you're sorry.
<sc>buy.com/BuyCar("BMW Z3")</sc></quote>
```

Thus, by sending an intensional result, the *qod* peer may force its clients to invoke dangerous services.

Finally, perhaps the most natural violation of security is to bring an AXML peer to transmit private data to a malicious distant site. This may be achieved for instance by including the following call (as a parameter of a call or in a result):

```
<sc>i.am.bad/SneakAbout([../../*])</sc></axml>
```

Instantiating this XPath argument amounts to sending i.am.bad (possibly private) parts of the document that included this call, which is clearly an issue.

The above examples show that the AXML framework makes unauthorized attempts to access data quite likely, as well as malicious usage of web services. Hence, access control is essential. We next see how this can be incorporated in the framework.

### 5.2 Our solution

We illustrate how the above issues may be addressed with two very simple policies. These policies have to be combined with some access control mechanism on the documents. Access models for XML have been proposed in, e.g., [16]. This aspect will not be detailed here.

In the first policy, called *binding*, a peer publishes the kind of arguments each of its services accepts (e.g., arbitrary AXML, XPath expressions, strict XML). Only calls with the proper arguments may then be activated. Note that this policy can be enforced using the WSDL language which enables publication of XML Schema types for services input/output parameters.

The second policy, called *trust*, reflects some form of agreement between the caller and the receiver. More precisely, the reasoning that allows to decide whether a service $sv$ (where $sv$ includes the name of the service and the site that provides it) can be called by a site $S$ is encapsulated in a boolean function $canCall(sv, S)$. The $canCall(sv, S)$ function returns *true* if $S$ is willing to call $sv$ *and* the provider of $sv$ is willing to accept this call from $S$. Note that, like in Java's sandbox security model [22], the decision depends on the origin of the call. This function will be used to determine which calls are eligible for activation at each point in time. We will see exactly how this is done in the next section.

To implement $canCall$, we can assume, for instance, that each peer has an *agreed service list*, containing the services that it trusts, and is willing to call. Similarly, we assume for every service, an *agreed site list*, i.e. the sites (trusted and accepted by the service provider) from which the service may be called. These two lists are typically exposed as web services. More precisely, each AXML peer $S$ provides (i) a service that allows to check whether it is willing to let another peer $S'$ call one of its services and (ii) a service to check whether it is willing to call some particular service. For non-AXML peers, we make conservative assumptions.

As mentioned above, these two models, *binding* and *trust*, may be combined. They may also be extended in a number of ways. First, one may want to include some access control list (ACL) mechanism, to grant different rights to various users of a peer. One might

want to control the right to fire a particular service call or the right to access data with an arbitrary granularity (e.g., at the element level). Also, the *canCall* function may vary in time. For instance, depending on the load of the service provider, one may want to restrict usage of the service to certain clients only. Finally, one may want to include arbitrarily complex solutions for trust management that have been proposed such as REFEREE [14]. No matter how complex the used policy is, the evaluator essentially needs to know, given a concrete call and a site, whether this site is entitled to activate this particular call.

## 6 Evaluation and Implementation

In this section, we describe the architectural components, and the algorithms used by an AXML peer in order to evaluate and maintain AXML documents. First, we explain how time-related events are detected in the system. Then, we see how the evaluation of documents is affected by these events.

**The Event Detector** To capture time-related events, we use an *Event Detector* module (ED). For simplicity, we omitted this module from the architecture sketch (Figure 1) at the beginning of this paper. The ED of an AXML peer $S$ monitors all AXML documents on $S$, including data validity parameters, and the activation mode and frequency of all service calls present in these documents. The ED sends messages to other components of the AXML peer:

- to the Evaluator: when a service call has expired, or has reached timeout;
- to the AXML storage: when a data node has become invalid.

Before presenting our evaluation algorithm, recall from Section 3.4 that service calls can be defined to be *immediate* or *lazy*. Immediate service calls have to be activated as soon as they expire, while the activation of expired lazy calls may be postponed until their results are actually needed. To simplify the presentation, we first assume below that all the service calls in the documents are defined with an immediate execution mode, and explain the evaluation algorithm for this restricted case. Next we explain how the above needs to be extended in order to support lazy calls. Finally we describe our implementation.

Recall from Section 4 that a *concrete* service call is one whose parameters do not include XPath expressions.

### 6.1 Calls with immediate mode

We start by explaining how the Evaluator decides when a call is *eligible* for instantiation, resp. activation, computation and return, (in the terms of Section 4), based on the messages received from the ED. Web then outline the algorithms for processing service call activations.

**Deciding on call eligibility** Let us consider the meaning of a "sc has expired" message.

- If sc is non-concrete, the message informs the Evaluator that sc is *eligible for instantiation.*
- If sc is concrete and aimed at some service outside $S$, we first choose some of the service calls included in the parameters (according to the security, capability, and optimization reasoning outlined in Section 5), and process them. Only then, sc becomes *eligible for sending.*
- If sc is concrete and aimed at a local AXML service defined on $S$ by a query $Q$, then sc becomes *eligible for computation.*

So far we considered eligibility for instantiation, sending and computing. For becoming *eligible for returning*, the service result may need to be post-processed (again, by calling some of the service calls in the result, based on security, capability etc.)

**Processing service call activations** Recall from Section 4 that for every service call activation on an AXML peer, a temporary document, that we call a *task*, is created in the working area $\mathcal{W}$. This document contains an sc-node corresponding to the <sc> element for which the task was created, and has a destination, that may well be a node in another task; thus, tasks are inter-connected by dependencies. We allow within the evaluation some non-deterministic, parallel behavior: at a given point in time, a task is either *ready*, or *suspended*, waiting for some event, perhaps the end of other task, and any of the ready tasks may be processed at that point. Thus, tasks in $\mathcal{W}$ are organized as a set of trees: the leaf tasks are all ready, and any of them may be chosen, while the non-leaf tasks are suspended. In the following, a *concrete task* is one that was created for a concrete call.

Tasks are created in three possible ways. First, the Evaluator creates a new task (concrete or non-concrete) for the activation of every expired, immediate service call. Second, upon receiving from outside a call to a service local to $S$, the SOAP wrapper creates a task for this call in $\mathcal{W}$. Note that this task is concrete, since only concrete tasks can be sent (see Section 4). Third, the processing of a task may create other tasks, as we will see.

As a notation, let $t(d, S_f, f, p_1, p_2, \ldots, p_n)$ be a task with destination $d$, corresponding to the activation of a call to the service $f$, provided by the peer $S_f$, with parameters $p_1, p_2, \ldots, p_n$; let $d.doc$ and $d.peer$ be the document, respectively the peer of the task destination.

Figure 2 outlines the simple algorithm for evaluating non-concrete tasks. First, the XPath parameters of the task have to be evaluated, by issuing calls to the query processor. When the evaluation is done, each $p_i$ has the value of an AXML forest $f_i$. As explained in Section 3.1, the non-concrete call is unrolled into as

| | peer $S$, non-concrete task $t(S_f, f, p_1, p_2, \ldots, p_n, d)$ |
|---|---|
| 1 | evaluate the XPath parameters in $p_1, p_2, \ldots, p_n$ |
| 2 | foreach $i = 1, 2, \ldots, n$ |
| 3 | let $f_i$ be the value of $p_i$ (an AXML forest) |
| 4 | foreach $x = (x_1, x_2, \ldots, x_n) \in f_1 \times f_2 \times \ldots \times f_n$ |
| 5 | create $t_x(S_f, f, x_1, x_2, \ldots, x_n, t.root)$ |
| 6 | insert $t_x$ in $\mathcal{W}$ |
| 7 | suspend until all $t_x$ finish |
| 8 | exit |

Figure 2: Processing a non-concrete task.

many concrete calls as there are elements in the cartesian product of the forests $f_i$. The processing of $t$ is over when all these concrete tasks have finished.

In Figure 3, we describe the processing of a concrete task. Assume that a parameter $p_i$, which is an AXML tree, contains some expired call to $sc$. Then, $S$ has to decide whether it needs to activate this call or send it as an intensional parameter. The decision is done based on the *binding* and *trust* policies described in Section 5. If both options are valid, the decision is made based on considerations like the systems load. Note that the decision is made locally, using the policies of $S, S_f, sc$ without requiring a "global" view of the security and capability requirements of other peers.

At line 6, if $f$ is a service local to $S$, then we call the XQuery processor with the proper arguments; otherwise, a call is sent to $S_f$ via the SOAP wrapper. In both cases, $t$ is suspended waiting for the result; and if $f$ is continuous, the activation also contains a "subscribe" request, and $t$ only waits for the first result to come. Once $S$ receives the result, if it needs to forward it to the distinct peer $d.peer$, we may have to decide when and where to execute the calls it contains. The reasoning is very similar to the one above, dividing the work among $d.peer$ and $S$. Subsequently, the result is sent to $d$. (If $d$ is local, by accessing the local AXML repository; otherwise, by sending a result message through the SOAP wrapper). Finally, the concrete tasks exits if $f$ is a non-continuous service; for continuous services, the task is kept, as it is the receiver of all the updates from $f$. Upon receiving the update, $t$ forwards it to be inserted under $d$, discards the data, and waits for the next update.

**Unsubscribe and timeout** For readability, we have omitted some issues from the algorithms depicted in Figures 2 and 3. First, if an unsubscribe message for a continuous service is sent by the ED, the Evaluator identifies the associated concrete tasks, sends an "unsubscribe" message to the service provider for each task, and destroys the task. Similarly, when a non-continuous call times out, the Evaluator destroys its task.

### 6.2 Calls with lazy mode

Let us now consider the more complex case of the lazy mode.

| | peer $S$, concrete task $t(S_f, f, p_1, p_2, \ldots, p_n, d)$ |
|---|---|
| 1 | foreach $sc_i$ in $p_1, \ldots, p_n$ |
| 2 | if $S$ decides to activate $sc_i$ |
| 3 | then create $t_i$, new task for $sc_i$; insert $t_i$ in $\mathcal{W}$ |
| 4 | suspend until all $t_i$ finish |
| 5 | if $S = S_f$ (i.e. $f$ is defined in $S$ by some query $Q$) |
| 6 | then call $Q(p_1, p_2, \ldots, p_n)$; suspend until result ready |
| 7 | else (i.e. $f$ is a distant service) |
| 8 | call $S_f/f(p_1, p_2, \ldots, p_n)$; suspend until result ready |
| 9 | if $S \neq d.peer$ |
| 10 | then foreach $sc_j$ in result |
| 11 | if $S$ decides to activate $sc_j$ |
| 12 | then create $t_j$, new task for $sc_j$; insert $t_j$ in $\mathcal{W}$ |
| 13 | suspend until all $t_j$ finish |
| 14 | send result to be inserted under $d$ |
| 15 | if $f$ non continuous |
| 16 | then exit |

Figure 3: Processing a concrete task.

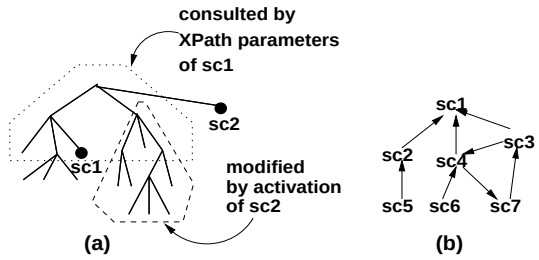


Figure 4: Dependencies among service calls.

**Service call dependencies** The presence of lazy calls may cause dependencies among call activations. For example, assume that we need to activate a non-concrete service call. Before instantiating its XPath parameters, we may need to activate some lazy service calls, that may affect the result of the instantiation. This situation is illustrated in the AXML document shown in Figure 4(a). The "influence zone" of $sc_2$, i.e., the set of nodes that may be modified by the results of $sc_2$, intersects the zone in which the XPath parameters of $sc_1$ are evaluated.

If $sc_2$ is in lazy mode, and has expired, then it is preferable to call it again before we instantiate the XPath arguments of $sc_1$. In turn, $sc_2$ may have XPath parameters that evaluate in the influence zones of lazy, expired service calls, leading to a graph of dependencies like the one in Figure 4(b).

Similarly, assume that a request for an AXML service is received and the service query $Q$ needs to be evaluated. Before calling the XQuery processor, we have to check if the data read by $Q$ intersects the influence zone of some lazy expired service call. This again leads to a dependency graph of the above form.

A reasonable compromise between precision and complexity has to be found for tracking dependencies. It is not possible to compute dependency graphs statically. For instance, as a document evolves, calls are added, or removed, by service call activations. Com-

puting the *exact* dependency graph of a service call leads to computationally complex problems such as XPath containment [18].

We therefore adopt the following pragmatic solution. We consider the influence zone of a service call to be all the subtrees rooted at its parent. We consider the scope of an XPath expression to be the set of subtrees rooted in the highest nodes attained by its evaluation, as described by the XPath specification [47]. Finally, we assume the data read by an XQuery query to be described by the XPath expressions in its for clause[5]. In general, path expressions may appear also in other parts of the query, e.g. the where clause. W.l.o.g we assume here that the query is first normalized [29]. We have thus brought the dependency decision problem to deciding whether two trees intersect, which can be done in constant time, provided a convenient encoding for element IDs (e.g., [28]).

A call dependency graph may contain cycles reflecting mutual call dependencies. They are broken by arbitrarily choosing some dependencies to be ignored. Breaking the cycles amounts to introducing non-determinism and possibly "missing" some data. In a web context, this is acceptable.

**Eligibility with lazy mode** In the presence of lazy calls, a given call sc may be declared eligible for instantiation (resp. execution) only after all the lazy calls in its data dependency graph have been issued.

**Call activation with lazy mode** Task processing in the presence of lazy calls is more complex due to the fact that we have to track data dependencies. First, before instantiating an XPath argument of a non-concrete call, we have to make sure that the data it bears on is available. To that purpose, before line 1 in Figure 2, we need to construct the dependency graph $G$ for the XPath parameters of the task, on a snapshot of the destination document. If $G$ has cycles, they are broken; then, we create tasks for all the leaf nodes from $G$, and process them in parallel. When these tasks are over, to take into account their effect on the destination document, we re-compute $G$; as long as $G$ is not empty, we repeatedly create and process tasks, corresponding to lazy, expired calls, that $t$ depends on. The processing of $t$ is suspended until $G$ is empty.

The very same steps have to be applied when processing a concrete task, before actually calling the XQuery processor (line 6 in Figure 3), except that $G$ is computed for the XPath expression that $Q$ depends on. We omit the details.

### 6.3 Implementation

A first prototype of AXML peer software has been implemented in Java. It relies on the XOQL query processor [5] which implements an algebra similar to the one of XQuery [6]. The SOAP wrapper, which is needed both to invoke and answer service calls is implemented using the Axis engine from the Apache software group [7], which although in early development stage, provides good performance and great flexibility through its architecture based on chainable handlers.

We implemented the evaluation strategy of Section 6.1, which only deals with immediate activation of service calls. The latters are scheduled by one or several timer threads. In this restricted case, dependency among service calls does not have to be tracked. The next version will support lazy calls as well.

To conclude this section, note that SOAP supports only RPC calls and one-way messages. For continuous services, we built a layer on top of SOAP [15], where the caller of a continuous service provides a listening service used by the callee to return a stream of answers.

## 7 Conclusion

The AXML paradigm allows to turn service calls embedded in XML documents into a powerful tool for data integration. This includes in particular support for various integration scenarios like mediation and data warehousing and distributing computations over the net via the exchange of AXML documents.

We implemented a first prototype but further work is needed to develop appropriate optimization techniques. Because of the richness of the model, this is a complex task that should borrow from many techniques that have been previously used, notably in the contexts of warehouses and mediators. We also need to build an environment for designing AXML documents and in particular, a graphical editor/browser for AXML documents.

The proposed AXML paradigm should be further experimented and evaluated. Towards this goal, we are intending to use AXML as an applications development platform in the context of a european project, namely DBGlobe. The project deals with data management problems in global distributed computing environments, with en emphasis on mobility. We believe it provides an adequate testbed for the proposed framework.

## References


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

[2] S. Abiteboul, O. Benjelloun, and T. Milo. A Data-Centric Perspective on Web Services (Preliminary Report). Technical Report 212, INRIA, November 2001.

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


[5] In some sense, this simple approach is pessimistic, since we do not use the where clause to filter the data actually consulted by $Q$.

[6] We chose XOQL because at the time we started this implementation, no XQuery processor was available to us. Although there are differences with XQuery, these are mostly syntactic.

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

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

[6] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. In *Proc. of ACM PODS*, 2001.

[7] The Apache Software Foundation. `http://www.apache.org`.

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

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

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

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

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

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

[14] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *Proc. of the Int. WWW Conf.*, volume 29(8-13), pages 953–964, 1997.

[15] F. Cremenescu. Supporting Subscription Services using SOAP, 2001. Stage de fin d'étude, Ecole Polytechnique.

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

[17] A. Deutsch, M.F. Fernandez, D. Florescu, A.Y. Levy, and D. Suciu. A Query Language for XML. In *Proc. of the Int. WWW Conf.*, volume 31(11-16), 1999.

[18] A. Deutsch and V. Tannen. Containment of Regular Path Expressions under Integrity Constraints. In *Proc. of the KRDB Workshop*, Rome, 2001.

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

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

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

[22] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. Proc. of the Usenix Symp. on Internet Technologies and Systems, 1997.

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

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

[25] H. Hosoya and B. C. Pierce. XDuce: A typed XML Processing Language (Preliminary Report). In *Proc. of WebDB*, May 2000.

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

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

[28] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of VLDB*, 2001.

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

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

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

[32] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of ACM SIGMOD*, 2001.

[33] Ozone: Integrating Structured and Semistructured Data. T. Lahiri and S. Abiteboul and J. Widom. In *Proc. Int. Workshop on Database Programming Languages*, 1999.

[34] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proc. of VLDB*, pages 413–424, 1996.

[35] J. Powell and T. Maxwell. Integrating Office XP Smart Tags with the Microsoft .NET Platform. `http://msdn.microsoft.com`.

[36] Simple Object Access Protocol (SOAP) 1.1. available at `http://www.w3.org/TR/SOAP`.

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

[38] I. Tatarinov, Z. Ives, A. Levy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

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

[40] J.D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[41] G. Weikum, editor. *Infrastructure for Advanced E-Services*, volume 24, no. 1. Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society edition, Mar 2001.

[42] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[43] G. Wiederhold. Intelligent Integration of Information. In *Proc. of ACM SIGMOD*, pages 434–437, Washington, DC, May 1993.

[44] Web Services Definition Language (WSDL). available at `http://www.w3.org/TR/wsdl`.

[45] Extensible Markup Language (XML) 1.0 (2nd Edition). available at `http://www.w3.org/TR/REC-xml`.

[46] XML Schema. available at `http://www.w3.org/TR/XML/Schema`.

[47] XML Path Language (XPath) Version 1.0. available at `http://www.w3.org/TR/xpath`.

[48] XQuery 1.0: An XML Query Language. available at `http://www.w3.org/TR/xquery`.

## APPENDIX 1: Deterministic computation and termination

We consider here monotone services, defined essentially with conjunctive fragments of XQuery/XPath, and assume that any piece of information has an infinite validity. We next briefly revisit the various steps of the computation and analyze what might be the sources on non-determinism in each step and what is required in order to avoid it.

**XPath expressions** The instantiations of an XPath expression may change in time because some nodes on the way have acquired more data because of the activation of some service calls. Thus the particular point in time in which the XPaths are evaluated may affect the result. To avid this problem, we must require that a call be re-instantiated whenever some new XPath instantiation occurs (i.e. using frequency="when param changes").

**XQueries** Similarly, the evaluation of a service query also change in time. (To be more precise, since we are in a monotone context, it may grow larger as time passes.) For continuous services, this is not a problem - the query is repeatedly evaluated and all new results are eventually sent. In contrast, non-continuous services are executed just once. Thus, we must require that the query be computed only after the data it bears on has be acquired. (We omit the formal definition of this notion here.)

**Exchanging data** Sites exchange data via call parameters and call result. The exchanged subtrees are extracted from a tree on one site and merged into a tree on the other side. Non determinism arises when the extracted subtrees contain path expressions that attempt to go above the root of the subtree: these paths have different instantiations if evaluated before or after the transfer. To avoid this, the exchange of such data is not allowed.

The above conditions guarantee that any data that can be derived in one particular execution sequence will also be eventually derived in any other sequence. Based on this, we have:

**Theorem .1** *For AXML documents and services satisfying the above conditions all computations lead to a unique state.*

However, this state may be finite for some AXML documents and infinite for others. Using a reduction of the halting problem for Turing machines, one can show that:

**Theorem .2** *Given an initial instance, one cannot decide whether it has a finite semantics.*

To conclude this section, we mention a particularly simple case where finite semantics is guaranteed.

**Layered services** A lot of the complexity comes from recursion and from the use of XPath expressions. In practice, services are often defined using layers, with services calling only services in lower layers, and returning data containing calls to services in such lower layers. We will call AXML documents without XPath expressions *XPath-free* documents, and AXML services that use only such document *XPath-free* services. One can show, that for XPath-free layered documents and services the semantics is finite. In this context, it is simple to design evaluation strategies that detect termination.

## APPENDIX 2: Extended XML syntax for service call elements

The following is a sample definition of an sc element:

```
<axml:sc
  endpointURL="http://xyz.com:80/soap/rpcrouter"
  nameSpaceURI="urn:xmethods-email"
  methodName="GetEmail" >
  <axml:params>
    <axml:param name="PersonName">
      <axml:xpath> ./@pname </axml:xpath>
    </axml:param>
    <axml:param name="CompanyName">
      <axml:xpath> ./../@name</axml:xpath>
    </axml:param>
  </axml:params>
</axml:sc>
```

As standard with XML, this definition is rather verbose. But typically, AXML developers will use GUIs and users will not have to deal directly with this syntax. (We do not have such an interface yet but are planing to develop one.)

To differentiate service calls from the rest of the standard XML data, and avoid naming conflicts, we use a specific XML *namespace* [30] for them (represented by the *axml* prefix in the example).

The attributes of the sc element provide the necessary information to issue the call, using the SOAP protocol.