Active XML: A Data-Centric Perspective on Web Services

S. Abiteboul INRIA and Xyleme
O. Benjelloun INRIA
I. Manolescu INRIA
T. Milo INRIA and Tel Aviv U.
R. Weber ETH Zurich

Abstract

We present Active XML (AXML, for short), a declarative framework that harnesses web services for data integration, and is put to work in a peer-to-peer architecture. It is centered around *AXML documents* which are XML documents that may contain calls to web services. The language allows both the specification of such documents and the definition of new web services based on them.

While documents with embedded calls were already proposed before, AXML is first to actually turn calls to web services embedded in XML documents into a powerful tool for web-scale data integration. In particular, the language includes linguistic features to control the timing of service call activations, the lifespan of data, and the exchange of extensional and intensional data. Various scenarios are captured, such as mediation, data warehousing, and distributed computation. A first prototype is described.

1 Introduction

Data integration has been extensively studied in the past in the context of company infrastructures e.g., [22, 45, 38]. However, since the web is becoming a main target, data integration has to deal with its large scale, and faces new problems of heterogeneity and interoperability between "loosely-coupled" sources, which often hide data behind programs. These issues have been recently addressed in two complimentary ways. First, major standardization efforts have addressed and partially resolved some of the heterogeneity and interoperability problems via (proposed) standards for the web, like XML, SOAP and WSDL [42]. XML, as a self-describing semi-structured data model, brings flexibility for handling data heterogeneity [1], while emerging standards for *web services* like SOAP and WSDL simplify the interoperability problem by normalizing the way programs can be invoked over the web. Second, the increasingly popular peer-to-peer architectures seem to provide a promising solution for the problems coming from the independence and autonomy of sources and the large scale of the web. Peer-to-peer architectures provide a decentralized infrastructure in sync with the spirit of the web and that scales well to its size, as demonstrated by recent applications, e.g., [31, 27].

[1] Complementary standards for meta data, like RDF, also address semantic heterogeneity issues.

We believe that the peer-to-peer approach, together with the aforementioned standards, form the proper ground for data integration on the web. What is still lacking, however, is the glue between these two paradigms. This is precisely what is provided by Active XML (AXML, in short), the topic of this paper, a declarative framework that harnesses XML and web services for data integration, and is put to work in a peer-to-peer architecture.

The AXML framework is centered around *AXML documents* which are XML documents that may contain calls to web services. When calls included in a document are fired, the latter is enriched by the corresponding results. In some sense, an AXML document can be seen as a (partially) materialized view integrating plain XML data and dynamic data obtained from service calls. It is important to stress that AXML documents are syntactically valid XML documents. As such, they can be stored, processed and displayed using existing tools for XML. Since web services play a major role in our model, we also provide a powerful mean to create new ones: they can be specified as queries with parameters on AXML documents, using XQuery [52], the future standard query language for XML, extended with updates.

Documents with embedded calls were already proposed before (see related work in Section 2 for a detailed account). But AXML is first to actually turn them into a powerful tool for data integration, by providing the following features:

Controlling the activation of calls and the lifespan of data By giving 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.

Services with intensional input/output Standard web services exchange "plain" XML data. We allow AXML web services to exchange AXML data that may contain calls to other services. Namely, the arguments of calls as well as the returned answers may include both extensional XML data and intensional information represented by embedded calls. This allows to delegate portions of a computation to other peers, i.e., to distribute computations. Moreover, the decision to do so can be based on considerations such as peer capabilities or security requirements.

Continuous services Existing web services either use a (one-way) messaging style or a remote procedure calls (RPC) style — they are called with some parameters and (eventually) return an answer. But often, a *continuous* behavior is desired, where a (possibly infinite) stream of answers is returned

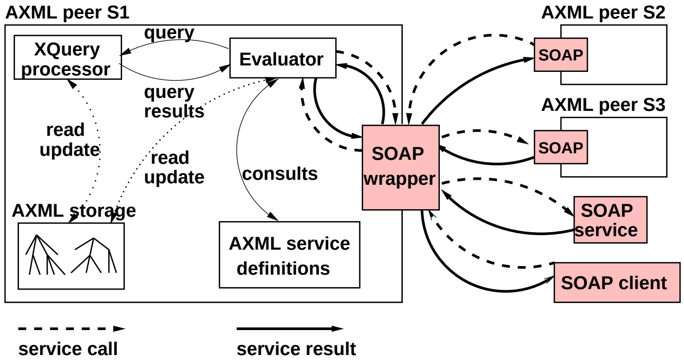

Figure 1: Outline of the AXML data and service integration architecture.

for a single call. As an example, consider subscription systems where new data of interest is pushed to users (see, e.g., [33]). 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 adds this capability to web services, and allows for the use and creation of such continuous web services.

These features are combined in a peer-based architecture, illustrated by Figure 1. Each *peer* contains a repository of AXML documents, some AXML web services definitions, and an Evaluator module, that acts both as a client and as a server:

Client By 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 complies with standard protocols for web service description and invocation [37, 46].

Server By 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. results), 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

Active XML touches several areas in data managment and web computing. We next briefly consider these topics and explain how AXML relates to them.

Basic technologies/standards The starting point of the present work is the semistructured data model and its current standard incarnation, namely XML [49]. We rely primarily on an XML query language and on protocols for enabling remote procedure calls on the web. Disparate efforts to define a query language for XML are now unifying in the XQuery proposal [52], and its subset XPath [51]. We use those as the basis of our service definition language. As for remote procedure calls, the various industrial proposals for web services, 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 [2]. Among those, we are directly concerned with the Simple Object Access Protocol (SOAP) [37] and the Web Services Description Language (WSDL) [46], that are used in our implementation. More indirectly, UDDI registries [40], can be used by our system, e.g., to publish or discover web services of interest.

Data integration Data integration systems typically consist of data sources, that provide data, and mediators or warehouses, that integrate it with respect to a schema. The relationship between the formers and the latters is defined using a global-as-view or local-as-view approach [20, 28]. While mediators/warehouses can also serve as data sources for higher integration levels, a clear hierarchy between the data providers typically exists. In contrast, all Active XML peers play a symmetric role, both as data sources and as partially materialised views over the other peers, thus enabling a more flexible and scalable network architecture for data integration. Web services are used as uniform wrappers for heterogenous data sources, but also provide generic means to apply various transformations on the retrieved data. In particular, tools developped for schema/data translation and semantic integration, e.g. [28] can be wrapped as web services and used to enrich the Active XML framework.

Services composition and integration Integration and composition of web services (and programs in general) have been an active fields of research [43]. Intensional data was used in Multilisp [23], under the form of "futures", i.e., handles to results of ongoing computations, to allow for parallelism. Ambients [11, 12], as *bounded* spaces where computation happens, also provide a powerful abstrac-

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

tion for processes and devices mobility on the web. More recently, standard languages have even been proposed in the industry, like IBM's Web Services Flow Language [47] or Microsoft's XLang [48] for specifying how web services interact with each other and how messages and data can be passed consistently between business processes and service interfaces.

While AXML allows to compose services, to use the output of some service calls as input to other ones, and even to define new services based on such (possibly recursive) compositions, the focus here is not on workflow and process-oriented techniques [14] but on *data*. AXML is not a framework for service composition, but for data integration using web services.

Our formal foundation is based on non-deterministic fixpoint semantics [4], that was primarily developed for Datalog extensions. In that direction, the paper has also been influenced by recent works on distributed Datalog evaluation [26].

Embedded calls As already mentioned, the idea of embedding function calls in data is not a new one. Embedded functions are already present in relational systems [41], e.g., as stored procedures. Method calls form a key component of object databases [13]. The introduction of service calls in XML documents is thus very natural. Indeed, external functions were present in Lore [25] and an extension of object databases to handle semistructured data is proposed in [34], 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 [36], where service calls can be embedded in Office documents, mainly to enrich the user experience by providing contextual tools. Our goal is to provide means of controlling and enriching the use of web service calls for data integration, and to equip them with a formal semantics.

Active databases and triggers The activation of service calls is closely related to the use of triggers [41] in relational databases, or rules in active databases [44]. Active rules were recently adapted to the XML/XQuery context [9]. A recent work considered firing web service calls [10]. 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 the automatic generation of applications where users could cooperate via data sharing and change control. The main differences with ActiveViews are that (i) AXML promotes peer-to-peer relationships vs. interactions via a central repository, and (ii) the cornerstones of the AXML language are XPath, XQuery and web services vs. object databases [13].

Peer-to-Peer computing is gaining momentum as a large-scale resource sharing paradigm by promoting direct exchange between equal-footed peers [27, 31]. We propose a system where interactions between peers are at the core of the data model, through the use of service calls. Morever, it allows peers to play different roles, and does not impose strong constraints on interaction patterns between peers, since they are allowed to define and use arbitrary web services. While we do not consider in this paper issues such as dynamic data placement and replication, or automatic peer discovery, we believe that solutions developped in the peer computing community for these problems (see for instance [31]) can benefit our system, and plan to investigate that in future work.

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 [3]. 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:

```
<myAuctions> Auctions I'm interested in.
 <category name="Toys">
  <sc>auction.com/getOffers("Toys")</sc>
```

[3]For readability, we use a simple syntax for <sc> elements.The complete syntax is presented in appendix.

```
 </category>
 <category name="Glassware">
   <sc>eBay.com/getAuctions("Glassware")</sc>
 </category>
</myAuctions>
```

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:

```
<myAuctions> Auctions I'm interested in.
 <category name="Toys">
   <sc>auction.com/getOffers("Toys")</sc>
   <auction aID="1"><description>Stuffed bear toy</description></auction>
   <auction aID="2">...
 </category>...
</myAuctions>
```

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., [35, 5, 18]. 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 ten currently most attractive offers, each one 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.

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 [51]:

```
<myAuctions> Auctions I'm interested in.
 <category name="Toys">
   <sc>auction.com/getOffers([../@name/text()])</sc>    ...</category>    ... </myAuctions>
```

The XPath expression ../@name/text() navigates from the <sc> node to the parent <category> element, and then to its name attribute. The service is then called with the name attribute's value as a parameter. In this example, there is only one possible instantiation for the XPath expression. In general, several document subtrees may match a given XPath expression. There are two possible choices here. Either to activate the service several times, once per each possible instantiation, or alternatively to call it just once, sending the forest consisting of all the instantiations a single parameter. In our implementation, we took the first approach as a default, but in principle one could add a attribute to the <sc> element to explicitly specify which one of the two semantics is prefered for a particular call. 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 default 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:

```
<myAuctions> Auctions I'm interested in.
 <category name="Toys">
   <sc>auction.com/getOffers1([../@name/text()], <sc>bank.com/getBudget("Bob")</sc>)</sc>
 </category>
</myAuctions>
```

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:

```
<myAuctions> Auctions I'm interested in.
 <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>...
</myAuctions>
```

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. Therefore, AXML service calls can be seen as a generalization of HTML hyperlinks, that handles calls to web services.

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 to manage these results, so that we may replace old results with new ones, discard data that became too old or 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 chose 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 [4]. Some nodes may have explicit expiration time, whereas others will inherit it from one of their ancestors. 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 also come with an 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, the following call states that auctions in Category A are archived for one year.

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

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

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 the service has to be called whenever the data produced by this call is needed (e.g., by a query over the AXML document).

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 be meaningful in this case).
- mediator style with caching: choose a non-zero value for valid, and lazy mode.
- warehousing mode with pulling information: 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>

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. An exception handling mechanism should also be provided to manage such events. □

3.5 AXML service definition

The AXML framework allows to call arbitrary web services, but also to define new ones, as illustrated in this section. In short, an AXML service is defined by a parameterized XQuery query 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 [50], as advised by the WSDL specification [46]. This information can be used to chose between the two options mentioned above.

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... Additional parameters can be defined to specify the frequency of updates, and whether to send full results or deltas. They are not detailed here.

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 [39].

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 [24, 7], static type checking becomes more complicated due to the use of ID-based element fusion and of XPath expressions in call parameters.

Termination and recursion We have seen above that a service call 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. Similarly, the processing of a particular service call (namely the evaluation of the query defining it) may triger the execution of new calls (perhaps even to the same service), etc. While some form of recursion is useful, e.g. for defining transitive closure type of computations, detecting termination in general is difficult due to the distributed form of the computation and the independence of the peers. 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. We will first present 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 $(\mathcal{R}_i, \mathcal{F}_i, \mathcal{W}_i)$ where $\mathcal{R}_i$, the peer's *repository*, is a set of persistent AXML documents, $\mathcal{F}_i$, the peer's *services*, is a set of AXML service definitions, and $\mathcal{W}_i$, the peer *working area*, is a set of AXML temporary documents. All the sets are assumed to be finite.

Each document d in the working area $\mathcal{W}_i$ 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 the place where the result of this computation should be sent, which can be a local element or a request from a remote peer.

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 [49] are that (1) we ignore here the order of elements, hence our trees are unordered [5], 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 becomes 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 $\langle p, f, x_1, \ldots, x_n \rangle$ where:

- p and f are respectively a *peer* and *service* names, or XPath expressions. In the first case, the service f must be defined in peer p with arity n.
- $x_1, \ldots, x_n$, the call parameters, are AXML documents or XPath expressions.

An sc-node where none of $p, f, x_1, \ldots, x_n$ are XPath expressions, is called a *concrete* call.

Reduced documents Continuous services send a sequence of answers to the caller. Smart (or optimized) 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.

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

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

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 $\mathcal{R}_i$ 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 $s = \langle \mathcal{R}, \mathcal{F}, \mathcal{W} \rangle$ 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 $\mathcal{R}$ 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 $\mathcal{R}$ 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 $sc\text{-}node$

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 $\mathcal{R}$ 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 the asynchronicity and the independence of peers, 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. Furthermore, assume that each call becomes eligible for instantiation and sending infinitly often (i.e. the call is repeatedly triggered and its XPath re-instantiated). 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. A finite state can be guaranteed with some additional restrictions. This is in the spirit of results on inflationary fixpoint semantics, see [4]. 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 receiver has the capability to process the call, which involves understanding the parameters that are sent. 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.

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 "Bob"'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., [17]. 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 [21], 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 [15]. No matter how complex the used policy is, the provider 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 P monitors all AXML documents on P, 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. Then, 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. We then outline the algorithms for processing service call activations.

Deciding on call eligibility The following rules are applied by the Evaluator module:

- Upon receiving an "sc has expired" message from the ED, if sc is non-concrete, it becomes *eligible for instantiation*.
- If sc is concrete and aimed at some service outside P, 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 P by a query Q, then sc becomes *eligible for computation*.
- After an sc aimed at a local AXML service was evaluated, its result becomes *eligible for returning* after being 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 the four steps of computation where chosen non-deterministically and in random order. We introduce here the notion of *task*, to track the evaluation of each particular service call, from the moment it is activated, to the end of its evaluation. Like sc-nodes, tasks can be *concrete* or *non-concrete*. Documents in $\mathcal{W}$ naturally have corresponding tasks, and so do activated sc-nodes in $\mathcal{R}$. Note that the evaluation is still non-deterministic, and that tasks can be evaluated in parallel: at a given point in time, a task is either *running*, *ready*, or *suspended*, waiting for some event, perhaps the end of another task. Any of the ready tasks may be processed at that point.

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

	peer P, non-concrete task $t(d, P_f, f, p_1, p_2, \ldots, p_n)$
1	evaluate the XPath parameters $p_1, p_2, \ldots, p_n$
2	foreach $i = 1, 2, \ldots, n$
3	let f_i be the value obtained for x_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 $t_x(P_f, f, x_1, x_2, \ldots, x_n, t.root)$
6	insert t_x in $\mathcal{W}$
7	suspend until all t_x finish
8	exit

Figure 2: Processing a non-concrete task.

outside a call to a service defined at P, 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 $t(d, P_f, f, p_1, p_2, \ldots, p_n)$ be a task with destination d, corresponding to the activation of a call to the service f, provided by the peer P_f, with parameters $p_1, p_2, \ldots, p_n$.

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 mentioned in Section 3.1, the non-concrete call is unrolled into as 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 service call sc. Then, P has to decide whether it needs to activate it or to send it as an intensional parameter. This decision is based on the *binding* and *trust* policies described in Section 5 [6]. Note that the decision is made locally, using the policies of P, P_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 P, then we call the XQuery processor with the proper arguments; otherwise, a call is sent to P_f via the SOAP wrapper. In both cases, t is suspended waiting for the result; Once P 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 P. 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.

[6] It may also take into account other considerations such as the system load.

peer P, concrete task $t(P_f, f, p_1, p_2, \ldots, p_n, d)$
foreach sc_i in $p_1, \ldots, p_n$
if P decides to activate sc_i
then create t_i, new task for sc_i; insert t_i in $\mathcal{W}$
suspend until all t_i finish
if $P = P_f$ (i.e. f is defined in P by some query Q)
then call $Q(p_1, p_2, \ldots, p_n)$; suspend until result ready
else (i.e. f is a distant service)
call $P_f/f(p_1, p_2, \ldots, p_n)$; suspend until result ready
if $P \neq d.peer$
then foreach sc_j in result
if P decides to activate sc_j
then create t_j, new task for sc_j; insert t_j in $\mathcal{W}$
suspend until all t_j finish
send result to be inserted under d
if f non continuous
then exit

Figure 3: Processing a concrete task.

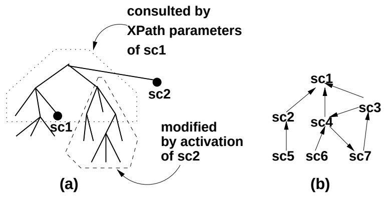

Figure 4: Dependencies among service calls.

Continuous tasks Tasks associated to calls to continous services keep running. The received updates just keep being sent to their destination. When they appear in the algorithms described above, these tasks are non-blocking.

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 "unsubscribe" messages to their service providers, 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.

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. Computing the *exact* dependency graph of a service call leads to computationally complex problems such as XPath containment [19].

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 [51]. Finally, we assume the data read by an XQuery query to be described by the XPath expressions in its for clause [7]. In general, path expressions may also appear in other parts of the query, e.g. the where clause. W.l.o.g we assume here that the query is first normalized [30]. 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., [29]).

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

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

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 expressions 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 [6] which implements an algebra similar to the one of XQuery [8]. The SOAP wrapper, which is needed both to invoke and answer service calls relies on the Axis engine from the Apache software group [8], 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 the immediate activation of service calls. This is done mainly using a timer thread that acts as a scheduler. In this restricted case, dependency among service calls does not have to be tracked. *Tasks* are evaluated in parallel, each one being handled by a separate thread. A thread pool mechanism is used to limit the number of simultaneously running threads.

Since SOAP supports only RPC calls and one-way messages, we built a layer on top of it to allow for continuous services [16]. Basically, the caller of a continuous service provides a listening SOAP service, used by the callee to return a stream of answers as one-way messages.

This prototype is functional, and was used to build a distributed peer-to-peer auctioning application, where each peer can offer auctions for other peers to bid on, and search for auctions of interest available from other peers, without needing a centralized auction server [3].

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, data warehousing and distributing computations over the web via the exchange of AXML documents.

[8] 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.

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 tools for easily building applications that use them.

The proposed AXML paradigm should be further experimented and evaluated. Towards this goal, we are intending to use AXML as an application development platform in the context of a European project called DBGlobe. The project deals with data management problems in global distributed computing environments, with a strong 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, O. Benjelloun, T. Milo, I. Manolescu, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration (demo). In *Proc. of VLDB*, 2002.

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

[5] 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.

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

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

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

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

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[42] The World Wide Web Consortium (W3C). `http://www.w3.org`.

[43] 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.

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

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

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

[47] Web Services Flow Language (WSFL 1.0). Available from `http://www.ibm.com/`.

[48] XLANG, Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.

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

[50] XML Schema. `http://www.w3.org/TR/XML/Schema`.

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

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

APPENDIX 1: Deterministic computation and termination

We consider here monotone services, defined essentially using 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 of non-determinism in each step and what is required in order to avoid it.

XPath expressions The instantiations of an XPath expression may change over time, because some nodes on the way may have acquired more data, due to the activation of some service calls. Therefore, the particular point in time when the XPaths are evaluated may affect the result. To avoid this problem, we must require that each non-concrete call becomes eligible for instantiation and sending infinitely often (i.e. that the frequency attribute is such that the call is repeatedly triggered and its XPath parameters are re-instantiated).

XQueries Similarly, the evaluation of a service defined as a query also changes over 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, because 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 been acquired. We omit the formal definition of this notion here.

Exchanging data Peers exchange data via call parameters and results. 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 only query such documents *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 (i) detect termination, and (ii) avoid re-triggering calls whose answer will not add new data.

APPENDIX 2: Extended XML syntax for service call elements

The following is a sample definition of a simple sc element that calls a corporate e-mail directory service:

```
<axml:sc
  endpointURL="http://xyz.com:80/soap/rpcrouter"
  nameSpaceURI="urn:xmethods-email"
  methodName="GetEmail" >
  <axml:params>
    <axml:param name="PersonName" xpath="../@pname" />
    <axml:param name="CompanyName">Microsoft</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* [32] 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.