Active XML: a data-centric view of web services

Serge Abiteboul
INRIA
serge.abiteboul@inria.fr

Omar Benjelloun
INRIA
omar.benjelloun@inria.fr

Tova Milo
INRIA & Tel Aviv University
tova.milo@inria.fr

Abstract

Web services provide a uniform access to data available on the web, from static pages, to form-based dynamically generated ones and most importantly, to complex applications with an external programmatic interface. A data-oriented approach to web services integration, namely Active XML, is introduced. The underlying model is based on XML documents possibly embedding calls to web services. When activated, these calls acquire more data that enrich the original documents. Web services operations are given semantics, and can be defined by means of (conjunctive) XQuery queries on Active XML documents. Being Active XML data themselves, the arguments and results of service calls may also contain service calls, which allows for distributed query processing over the web.

Documents with embedded calls to web services are already being used in commercial products, such as Microsoft Office, through its Smart Tags feature. In our contribution, we turn this concept into a powerful tool for data integration, and provide formal semantics for it. By offering linguistic features in the language that control the activation of calls, various data integration scenarios are captured, such as mediation and data warehousing. The model also takes into account web services that are based on possibly infinite streams of information such as those generated by query subscriptions.

A formal foundation based on non-deterministic, inflationary fixpoint semantics is presented for monotone services. A first implementation 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, a lot of efforts have been devoted to extending these techniques for the management of distributed data as well, and in particular to its integration, e.g., [19, 35, 20]. The web, and standards such as XML [10] (for data model) and SOAP [9] or WSDL [14] (for web services) are dramatically changing the context. XML provides self-describing semistructured data, while web services enable seamless access to static documents and to complex applications. Most importantly, by allowing applications to expose some of their functionality, independently from the platform they run on, or their technical architecture, these standards are opening a world of possibilities for data integration. However, a general framework and formal foundations are still missing to fully benefit from this new context.

The present paper is one step in that direction. Its main contribution is an XML-based language, namely *Active XML* (AXML, in short), that allows to interpret some XML elements as declarative calls to web services. AXML documents contain data, but also service calls that can be activated to acquire more data. The language also allows to define web services — and even actually to implement them — as queries, with parameters, on AXML documents.

Data with embedded calls to operations is an old idea. Indeed, it is already present in the context of XML and web services. In Microsoft Office XP, Smart Tags within XML documents can be linked to Microsoft's *.NET* platform for web services [23]. 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. Our approach lets users benefit from the uniform vision of a very wide range of heterogeneous sources provided by web services. It also allows to capture different kinds of data integration scenarios. Indeed, an essential question in this framework is when to actually invoke each particular service call. Based on the specification, the system may choose to do so when the information is needed (as in mediator systems), or beforehand (as in data warehouses). By allowing declarative control of service calls activation, the language we propose combines both approaches in a very flexible manner.

It is straightforward in AXML documents to use services that take XML data as arguments and return XML data. Beyond such simple web services, AXML data may refer to more complex services, which really allows to take advantage of the XML/web service technology:

Stream services: New services (e.g., subscriptions [24, 13, 21]) are emerging that entail a possibly permanent relationship between the caller and the service. Such services provide the caller with a (possibly infinite) stream of answers/notifications. As we will see, our framework allows to use such services in a variety of ways. It allows, for instance, to include in an AXML document, in a cumulative manner, the answers to particular query subscription. It also allows to use stream services to propagate changes from external sources to an AXML document.

Services exchanging AXML: Standard web services typically exchange "plain" XML data. In contrast, we allow web services to exchange AXML data that may contain calls to other services. This is in the spirit of [28] where the answer to a query is allowed to include both extensional and intensional information. In our framework, the arguments of a call may also contain service calls (intensional information). This allows, for instance, for delegating portions of a computation to other sites, i.e., to distribute computations.

The paper describes the linguistic features of the AXML language. In the presentation of the language, the emphasis is put on the integration of data and services. A formal foundation is proposed for a restricted framework, where service calls may only add new information, and do not invalidate existing one. The underlying data model is roughly XML without order (unordered labeled trees). Services are modeled using parameterized, monotone, XQuery-like queries. The semantics of AXML documents is based on non-deterministic inflationary fixpoint logic [2]. Some theoretical results are obtained such as the correctness of the semantics and undecidability results. We also discuss issues that arise when non-monotone computations are introduced.

We briefly describe a first on-going implementation of an AXML server. It relies on SOAP/WSDL for the web services. It is developed in Java using mostly open-source software from Apache [4] and the (also open-source) XOQL [3] query processor. AXML servers may be used in a wide variety of contexts, e.g, to process personal data in a PDA or to integrate large volumes of data in a warehouse. Therefore, the architecture we propose is very modular. Functionalities can be easily added/removed. Also, different implementations may be used for the same functionality, depending on the context. Our first implementation is targeted at small to medium size documents. It does cover the various features of the language.

Related work The starting point of the present work is the semistructured data model and its current standard incarnation: XML [10]. Disparate efforts to define a query language for the latter are now unifying in the XQuery proposal [12], that is used here for modeling web services as parameterized queries. 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) [9] and the Web Services Description Language (WSDL) [14], which are used in our implementation. More indirectly, UDDI registries [29], can be leveraged by our system, to find 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 [30], e.g., as stored procedures. Method calls form a key component of object databases [11]. The introduction of service calls in XML documents is thus very natural. Indeed, an extension of object databases to handle semistructured data is proposed in [25] 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. 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 clearly related to the use of triggers [30] in relational databases, or rules in active databases [34]. Some recent work has extended the latter for XML databases with trigger conditions expressed in XQuery and firing web service calls [7]. We harness these concepts to obtain a powerful data integration framework based on web services. More directly, the present work has been strongly influenced by previous works on *ActiveViews* [1]. In ActiveViews, declarative specifications allow the automatic generation of applications where users can cooperate via data sharing and control of change propagation. The main differences with ActiveViews are that (i) AXML promotes peer-to-peer relationships vs. relations via a central repository in ActiveViews, and (ii) the cornerstone of the AXML language is XPath, XQuery and web services vs. object databases [11] in ActiveViews.

Integration and composition of web services has been a very active field of research recently [32]. However, most of the works have focused on process-oriented techniques, and on the definition of workflows among services [15]. The present work is concerned with the complementary data-oriented aspects of web services integration.

The formal foundation is based on non-deterministic fixpoint semantics [2] that was primarily developed for datalog extensions. In that direction, the paper has also been influenced by recent works on distributed datalog evaluation [28].

Organization The AXML language is presented in Section 2, mainly through an extended example. In Section 3, we introduce the formal semantics of AXML documents. The first prototype is discussed in Section 4. The last section is a conclusion.

2 Active XML

In this section, we introduce *Active XML*, AXML in short. We first present the basic features of the language, focusing on data integration. So, in some sense, this first portion of AXML is in the spirit of languages for data integration, such as MSL [22]. We next consider more advanced aspects of the language that (i) allow to control data changes and (ii) permit the use of more complex services, producing possibly infinite streams of results. The formal semantics of AXML will be considered in the next section.

An AXML document is primarily an XML document. To be more precise, we partly ignore the ordering between the children of elements. In other words, we think of XML trees as unordered.

Clearly, the ordering exists in extensional data. However, when a service returns an answer or when a query is invoked, it may ignore the ordering of the elements in the result. XML documents may contain some particular elements tagged *sc* for *service calls*. In general, a web site may provide several web services. Each service may support an array of functions. We use the terminology *service call* for a call to one of the functions of a service.

We can think of the functions supported by a (web) service in many different ways. Such a function may be viewed as a method call (a SOAP [9], XMP-RPC [37] or Corba [27] call), that is given arguments and returns data. One can also think of it as a mean to wrap a form on the Web that has to be filled (with arguments) and submitted in order to get a new web page (returned result). Finally, one can see it, the approach we follow, as a query with arguments, where both the arguments and result are composed of AXML data. To specify the meaning of the functions supported by a service, we use the rising standard for XML querying, namely XQuery [12]. So for us, there is no difference between such a function and a query with parameters. This said, a function provided by a web service may be used without actually knowing its definition in XQuery. Indeed, the function may actually be implemented as a Java program or using any kind of code.

2.1 Data and simple service integration

To illustrate the language, we use the following simple example. A company is organized in departments. Each department has its own database of members and each member has its own personal database, e.g. stored in her Personal Digital Assistant (PDA). For us, all databases are (A)XML documents. We are concerned with building a view with all this information. Typically, the view will be used within the company to find accurate and up-to-date information about people, such as their e-mail address, phone, agenda, etc. We will first assume that the view is fully materialized and will remove this assumption further on. Such applications are standard and very typical of information sharing problems in an environment where data changes all the time (people move, their addresses and e-mail change). Clearly, the application is simplified here for the purpose of presentation.

The view we are defining is called *directory* and resides on a site called xyz.com. First, there are three departments as specified in the following definition:

```
<axml id="directory">
<dep name="Dvd"/> <dep name="Toy"/> <dep name="Bike"/>
</axml>
```

One can think of this data as forming a document, whose URL is xyz.com/directory.axml.

The view also includes information about department members. This information exists in department servers and is provided, for each department, by a particular service call. This can be syntactically specified with:

```
<axml id="directory">
  <dep name="Dvd"><sc>sesame.xyz.com/GetPersonel()</sc></dep>
  <dep name="Toy"/><sc>toy.xyz.com/GetPersonelToy()</sc></dep>
  <dep name="Bike"/><sc>bike2000.com/GetPersonnelBike()</sc></dep></axml>
```

So, for instance, the list of members of the toy department is provided by the server "toy.xyz.com". On that server, the function "GetPersonelToy" returns a list of all the department members as an XML document.

Although using web services is relatively easy, it does involve more than just giving the name of a service and the name of a function. One has also to deal with notions such as the namespace URI, etc., as defined, for instance, by the SOAP specification [9]. In AXML, these are defined using attributes of the *sc* elements. We will ignore these aspects here and use "logical services", simply defined by a site and a function name, thereby assuming that the mapping to *real* services is defined elsewhere. So, our setting consists of a set of sites. Each site includes a number of documents and some service functions.

Signatures are assigned to the functions of a service. A signature specifies the arity of the function, and/or names for its parameters and possibly DTDs (or XML Schemas) for the arguments and the result[1]. We showed above how to use the *GetPersonelToy* function (and similarly for the others). The function signature may be given by the following type definition, which, in particular, states that *pname* and *id* are identifiers for *person* and *pda*, respectively:

```
let sc toy.xyz.com/GetPersonelToy()
return (person*)
<!ELEMENT person (phone, pda?) > <!ATTLIST person pname ID REQUIRED >
<!ELEMENT phone (#PCDATA) >
<!ELEMENT pda (#PCDATA) >        <!ATTLIST pda id ID REQUIRED >
```

The definition of the function itself may, for instance, be given by:

```
let sc toy.xyz.com/GetPersonelToy() be
  for $a in document("toy.xyz.com/members.axml")/member,
      $b in $a//name, $c in $a//phone, $d in $a//pda
  return <person pname={ $b/text() }> { $c } { $d } </person>
```

Once these service calls have been performed, the data they return becomes part of the *directory* document, which may now look as follows:

```
<axml id="directory">...
  <dep name="Toy"/>
   <sc>toy.xyz.com/GetPersonelToy() </sc>
   <person pname="smith">
    <phone>01 32 32 32 32</phone>
    <pda id="smithPDA"><sc>alpha.xyz.com/smith_pda()</sc></pda> </person>
   <person pname="dupond">
     <phone>01 32 32 32 33</phone>
     <pda id="dupontPDA"><sc>toy.xyz.com/dupond2234()</sc></pda></person>
      ... </dep> ... </axml>
```

Observe in the example that the *pda* elements contain function calls. We will see more examples of this and discuss this important aspect of the language further on.

The functions that we defined so far have no input arguments. We next consider functions with arguments. To see an example, suppose that e-mail addresses may be obtained from a function, namely *GetEmail*, that receives the name of a person and her department and returns her e-mail address as an XML element of the form `<email>...</email>`. (We omit below the internal specification of the function, i.e. the associated query.) We can introduce in Smith's person element a call to this particular web service:

```
 <person pname="smith">
   <phone>01 32 32 32 32</phone>
   <pda id="smithPDA"><sc>alpha.xyz.com/smith_pda()</sc></pda>
   <sc>xyz.com/GetEmail("smith","Toy")</sc>
</person>
```

[1]In the web services context, this is done using the Web Services Description Language, in a nice, layered, though rather verbose syntax, that relies on XML Schema to express parameter types. For briefness, we will use a shorter DTD-based syntax in our example. Note however that our implementation relies on WSDL.

A call to the service allows to include Smith's email in the *directory* document. However, we may not want to repeat this specification explicitly for each person in the database. Instead, we can use a local web service to globally add this specification for each person element. For that, we add to the root element a service call to a function *AddEmail0()*,

```
<axml id="directory"> <sc>xyz.com/AddEmail0()</sc> ... </axml>
```

This function can be defined at the same site and specified as follows:

```
let sc xyz.com/AddEmail0() be
  for $a in document("xyz.com/directory.axml")/dep, $b in $a/person
  return
   <dep name={ $a/name }>
     <person pname={ $b/@pname }>
       <sc>xyz.com/GetEmail({ $b/@pname }, { $a/@name })</sc>
     </person> </dep>
```

This illustrates a subtlety of the integration of function results into the document. Recall that in XML, a DTD or XML Schema may specify that certain attributes uniquely identify their elements, i.e., that they are keys for these elements. By default, such returned elements with identifiers are merged with existing elements based on their identifiers, provided that their respective ancestors can be matched. Once this function has been activated, each person has an email element that contains a service call. Once the service call for Jones has been invoked, his/her e-mail address is recorded in the email child of Jones's element.

Note that the data model we are using is more complex than, say, the relational model. The next example illustrates some of the price we pay for this increase in modeling capabilities. As we will see in the example, some documents may become inconsistent, due to some data returned by the calls.

Example 2.1 (inconsistency) We assume that the document we start with is well-formed and indeed, that it is valid if a DTD (or XML Schema) is specified for it. An inconsistency may arise if one call leads to constructing a document that is not anymore well-formed or valid. An inconsistency may, for instance, occur if the same identifier is assigned to two distinct nodes in the document (which cannot naturally be merged). For example, suppose that we have the following document a in source s:

```
<a ID="hhh"><sc>s/G()</sc></a>
```

Suppose that $s/G()$ returns `<b> <a ID="hhh"/> <b>`. Then we have two distinct nodes (the parent and the child of the b node) with the same hhh identifier, which is of course an inconsistency.

Another possible type of inconsistency is related to attributes. Suppose that we know that an attribute a of an element has a value, say 5, and suppose a web service tells us that this attribute has value 6. Since attributes should have a unique value, this may be viewed as an inconsistency. (If the value comes from the same function call, we interpret the new value as an update of the old one.)

In many services found on the web, arguments and results of service calls are strings, although protocols like SOAP allow the use of richer types of arguments/result, and in particular XML data. In our AXML model, arguments may be arbitrary AXML data and even XPath expressions. We next examine these two cases.

Function calls in arguments and result In one of our previous examples, we already saw a function with a return value containing some function call (for the *pda* element). Note that services that return such intentional data already exist in real life. For instance, consider the Google search engine. Given some keywords, it returns a list of documents containing these words, plus (possibly) a handle for obtaining more answers. With this handle, one can obtain a new list and (possibly) yet another handle. In AXML terminology, a call to Google, say *Google/search("axml software", "1", "10")* may be interpreted as returning the following answer:

```
<answer>
<query from="1" to="10">axml software</query>
<res> 1) ... </res>  ... <res> 10) ... </res>
<more><sc>Google/search("axml software","11","20")</sc></more>
</answer>
```

Thus, our formalism nicely captures hyperlinks to both static and dynamic web pages, by modeling them as service calls, and the pages they return as AXML documents.

The following example shows that the possibility to return service calls in results is a potential cause of non-termination.

Example 2.2 (recursion) Assume that we have the following document a in source s:

```
<a> <sc>s/F()</sc> </a>
```

Suppose that a call to $s/F()$ returns `<a> <sc>s/F()</sc> </a>` as an answer. We are lead to the following sequence of document states:

```
<a><sc>s/F()</sc></a>
<a><a><sc>s/F()</sc></a><sc>s/F()</sc></a>
<a><a><a><sc>s/F()</sc></a><sc>s/F()</sc></a><sc>s/F()</sc></a> ...
```

Note that we will never compute the "complete" document.

There may also be other causes on non-termination. For instance, a document a on a site may contain a call to a function on another site that uses some document b. Document b, in turn, may contain some call to a function on the first site that uses the document a. Then, an access to the document a (e.g. for evaluating some query) may lead to each site waiting for an answer from the other one. We will address the issue of termination in more details in Section 3.

So far we have seen examples of functions that return data containing function calls. Similarly, functions can also take, as input arguments, AXML documents containing service calls. We shall see an example of that in Section 2.3.

XPath expressions as argument Besides AXML documents, the language allows to use XPath expressions in service calls. XPath [16] is a standard regular path expression language for navigation within XML documents. Starting from a particular node (elements or attributes), an XPath expression returns a set of nodes. To illustrate the use of XPath, we present an alternative way of specifying the e-mail address of each company member. This can be achieved (in what we believe to be a more appropriate manner) using the following function definition:

```
 let sc xyz.com/AddEmail() be for $a in
document("xyz.com/directory.axml")/dep, $b in $a/person return <dep
name={ $a/name } > <person pname={ $b/@pname }>
<sc>xyz.com/GetEmail([./@pname],[../@name])</sc> </person> </dep>
```

The intuition is that, starting from the *person* element, in order to find the first function argument (i.e. the person's name), one has to go to its *pname* attribute. And similarly for the second argument, one has to move first to the person's parent element and then to its *name* attribute. The use of path expressions is an important feature of the language - it allows to install a service call in some element of an AXML tree even when its arguments are not yet known, i.e., may have to be obtained using other services. Note that an XPath expression returns a set of solutions. The instantiation of the path expressions may thus entail the creation of several concrete service calls, one per each instantiation.

We used XPath expressions above to describe the arguments of a service call. Similarly, one can use them to specify the server name and/or the function to be invoked. This is illustrated in the following example. Assume that we want to query a number of search engines, and ask each one for the top 10 pages containing the keywords "axml software", a standard functionality of meta search engines. The site names and the functions that need to be called for each search engine are described in the *search* elements of the following document. The path expression in the service call refers to them:

```
<doc> <servers>
<search>www.google.com/Google</search>
<search>www.altavista.com/AVi</search> ...  </servers>
<query id="axml_search">
<sc> [../search]("axml software","1","10") </sc>
</query> </doc>
```

From the *query* node, ".." takes to the *server* node, and "/search" takes to *search* nodes. For each instantiation of the XPath expression, we get one concrete call, so this would essentially behave as if the *query* node was defined as:

```
<query id="axml_search">
<sc> www.google.com/Google("axml software","1","10") </sc>
<sc> www.altavista.com/AVi("axml software","1","10") </sc> ...
</query>
```

Before moving to more complex features of the language, let us summarize what we have achieved so far. We can define a document that integrates data coming from several XML databases. Part of the document is defined extensionally and part intensionally using service calls. The semantics of the functions can be defined using parameterized queries. The function arguments and answers may themselves contain function calls. We ignored so far a number of issues. Most importantly, we did not explicit when the service calls in a document are activated. If all service calls are performed immediately and data is inserted in the document, this leads to a warehousing approach. If the service calls are activated only on demand to answer queries on the document, it is more like a mediation. We consider next more complex features of the language that allow (i) to control the activation of service calls (Section 2.2) and (ii) to take advantage of stream services such as query subscriptions (Section 2.3).

2.2 Controlling the activation of calls

In this section, we show how to specify some control of the activation of service calls. To do that, we use two particular attributes, namely *sc-mode* and *sc-valid*.

The *sc-mode* attribute may take two values, *lazy* or *immediate*. (The default is *lazy*.) In *lazy* mode, we wait until the related piece of data is needed before activating the call, while in *immediate* mode, we activate the call whenever the corresponding data is not valid anymore. Data validity is controlled using the *sc-valid* attribute, which specifies for how long some retrieved data remains valid. (Its default is *always*.)

To illustrate these concepts, let us complete the definition of e-mails in the *directory* view:

```
let sc xyz.com/AddEmail() be
  for $a in document("xyz.com/directory.axml")/dep, $b in $a/person
  where $b/status/text()="permanent"
  return <person pname={ $b/@pname }>
           <sc sc-valid="1 year" sc-mode="immediate">
               xyz.com/GetEmail([./@pname],[./../@name])</sc></person>
```

The *immediate* mode indicates that the system should fetch the email addresses of permanent employees without waiting for it to be requested. Once an email has been acquired, it remains valid for one year. After one year, as specified by the *sc-valid* attribute, it is considere stale and the system should immediatly refetch it (because of the *sc-immediate* mode). For people who are not employees, we may want to choose the *lazy* mode. The activation of the appropriate calls is achieved by (virtually) attaching some additional attributes to each piece of data, recording the necessary information. Each piece of information that is obtained from service calls comes equipped with attributes such as *source* (the server that produced this particular subtree) and *acquisition_time* (acq_time), i.e. the time the information was produced. In the previous example, *sc-valid* and *sc-mode* were defined explicitly in the view. However, it is possible to let the responsibility of defining some of these attributes to the sources. For instance, we may let the source define the validity time for non permanent employees.

Note that, in the above example, the value that is returned by the *GetEmail* service call (namely the e-mail address) is a string, i.e., a PCDATA. Strictly speaking, it is not possible to attach an attribute to such a node. It is possible in AXML to do so. Details are omitted. We now have the necessary machinery to adopt various styles of data integration:

- mediator style: by simply fixing the validity time to 0. (Note that an immediate *sc-mode* would not have any meaning in that case).
- mediator style with caching: by choosing a non 0 validity time and lazy mode;
- warehousing mode with pulling information: by choosing a non 0 validity time and a immediate mode.
- warehousing mode with pushing: this will be addressed in the following subsection using a last feature: *streams of results*.

Attributes such as *sc-mode* are used to control changes. Considering change control immediatly leads to notions such as workflow, concurrency and transactions. Although these are definitly interesting directions of research, they will not be pursued here.

2.3 Streams of results

So far, the services we considered interact with an AXML document in a very simple manner. They are called with parameters and they (eventually) return an answer. There are situations however where, rather than sending a single answer, the service provides a (possibly infinite) stream of answers. Such streams of results may come, for instance, from sensors (e.g., thermometers) or monitors (e.g., video surveillance). In a database environment, they may arise from warehouse maintenance - the data sources send to the warehouse streams of messages consisting of the updates that occurred [33]. Another example is subscription systems - a user may request to receive all the wires from a press agency about soccer. Finally, a continuous query [13], i.e., a query that is evaluated with a certain frequency, e.g., weekly, also yields a stream of results.

Such services can be used in our setting, and indeed were one of the early motivations of the present work. We first consider a "cumulative" use of a stream of answers, that typically captures query subscriptions. We then consider a "non-cumulative" use that leads to the warehousing with pushing mentioned in the previous section.

To illustrate, consider for instance a subscription system where one may ask to be notified each time the system discovers a new document containing some specific keywords. For instance, such a service is supported by [38, 21]. Let us assume that the function providing this service is called XWatch. The following example demonstrates a possible use of XWatch. Assume the *directory* document also contains a call to the following function:

```
let sc PersonelOnTheWeb() be
    for $a in document("xyz.com/directory.axml")/dep, $b in $a/person
    return
     <dep name={ $a/name }>
       <person pname={ $b/@pname }> <sc mode="immediate">
         xwatch.com/XWatch("contains" || [./pname/text()] ||
         "and contains 'xyz' and new") </sc></person></dep>
```

The activation of *PersonOnTheWeb* at the root of *directory.axml* results in introducing a function call to XWatch in each *person* element. When we activate the function call, say for Smith, we register in the XWatch system a query subscription that requests to be notified of all new URLs that mentions "Smith" and "xyz" (the name of the company). The (infinite) stream of results from this subscription is inserted in a *watch* sub-element of Smith's *person* element. Assume that each result is an element of the form `<watch>...</watch>`. After a few notifications, the general structure of the data may look like:

```
<axml id="directory"> ...
  <person name="Smith">  ...
     <sc mode="immediate"> xwatch.com/XWatch(...) </sc>
     <watch>...</watch> <watch>...</watch> ... </person> ... </axml>
```

Intuitively, we opened a connection and started recording a flow of data coming from the service.

Now suppose that we want to use a *Filter* function provided by some site *f.com* that filters this flow of pages, e.g., to remove pornographic material. Then we could use in *PersonelOnTheWeb*

```
f.com/Filter(<d><sc>xwatch/XWatch("contains"...and new"</sc></d>))
```

The AXML local server will call *f.com/Filter* and pass it as argument a d element containing the *XWatch* call. Then Site f will call *XWatch*, filter its result and return the (flow of) filtered answers to *directory*. Note that this evaluation is possible only if (i) *Filter* accepts arbitrary AXML documents as input, and (ii) if *Filter* itself is a continuous service. An alternative evaluation strategy is to have the *directory* document call *XWatch* and call *Filter* with a stream of argument. This is in the style of pipelining. To simplify, we will consider here streams in results, a need in our context, and ignore the issue of infinite streams in function arguments.

The *XWatch* example illustrated how we could integrate a flow of data coming from an external source by simply cumulating it at the receiving node. This *cumulative* approach simplifies the underlying foundation: we remain in a monotone framework with some inflationary semantics. Now suppose that we have a continuous version of GetEmail that sends a new e-mail address each time the latter is updated. This allows us to have fresh information without having to poll service calls, i.e., e-mail addresses are maintained in a push mode. From an implementation viewpoint, this is exactly like a query subscription. We receive a flow of answers. The only difference is in the interpretation of answers: we want each new e-mail address for a person to invalidate the previous one. So, for instance, suppose that a *ContinuousGetEmail* function is available. We may request to be constantly updated on changes of the e-mail address of Smith:

```
<person pname="smith">
  <sc sc-valid="last">
    xyz.com/ContinuousGetEmail("smith","Toy") </sc> </person>
```

The value of the *sc-valid* attribute specifies that only the last received e-mail address should be considered as valid. Observe a difference with the management of e-mail using standard queries. Here, e-mail information is pushed to us by the service. We need to call the function only once to subscribe to it.

The definition of continuous functions is similar to the one of regular ones - the only difference is the special keyword *continuous* used at the function declaration. This definition may also include parameters to specify how often the notifications should be sent, e.g., weekly, and the style of updates, e.g., resending the entire value vs. simply sending a delta.

To conclude this section, observe that by removing the *sc-valid* attribute (with value *last* in the example), we can specify that all e-mail addresses should be kept. The default semantics is cumulative, and each result has an attribute that specifies when it was generated (see Section 2.2). This leads in some sense to versioning of data.

2.4 Discussion

We will present in the next section a formal semantics and results for the case of cumulative data acquisition, i.e., where data that is acquired is never invalidated. We describe this semantic here informally and discuss problems coming from non-monotonicity. We also mention the issue of evaluation.

The framework is by definition asynchronous because it models independent computation at autonomous sites. A site contains some AXML documents. When it receives a service call, it creates a new document, to hold the computation needed for this function, that may last for a while. The only difference from a semantic viewpoint with a “normal” document is that this document terminates when the corresponding computation terminates.

Let us first consider the cumulative case. This is achieved by limiting the queries to monotone queries and assuming that any piece of information has an infinite validity. If all functions are stream functions, we can define a precise semantics based on a nondeterministic inflationary fixpoint. At each step, one decides (nondeterministically) to perform some operation such as activating a service call or returning a result. The operation can only add information to the state. We shall see that, because of the monotonicity, the (possibly infinite) state that is eventually reached is independent of the order of activation of the operations, assuming some restricted fairness in the choice of operation activations. Next, we will consider non-stream (still monotone) functions and show that a precise semantics can also be obtained following analogous principles.

Even in the monotone case, termination is an issue. A function may produce an infinite result and so, its computation may never terminate since at each state, we only have finite documents. Even if the computation terminates, it is not easy in general to detect termination since, in our context, termination is a global property. For instance, one can implement any recursive datalog program [2] by having each intensional predicate at a different site (or each magic predicate at a different site if we use Magic Set [6] as a query evaluation strategy). Clearly, the computation terminates when all sites are simultaneously stable and some extra machinery is needed to detect termination. Nevertheless, we will see some reasonable conditions that guarantee termination.

Consider now the non-monotone case. Non-monotonicity may arise for various reasons. First, the extensional part of the data may be updated. For instance, the phone number of Smith may be deleted in the department server. As a consequence, the intentional part of the data may change. Another possible cause for non-monotonicity are functions defined by queries with negation. For instance, one may provide a service that returns the list of people for whom our directory knows no phone number. The value of this function may change in time in a non-monotone way. In general, this results in having to propagate non-monotone changes to documents that may have used the information that has changed. This leads to standard consistency problems in warehouse maintenance [33]. Not surprisingly, it is difficult to guarantee consistency in such a setting. Note that we are used to live with inconsistencies in a web context. For instance, consider a query to Google [18] with some keyword w. If an URL u is returned, this means that page u contained at some point (when it was last crawled) the keywords in w. There is no guarantee that it still contains them or that it even exists. Still, people use (and like) Google. Clearly, such (lack of) semantics is not acceptable for applications requiring a precise semantics. One might consider using strict concurrency control (a somewhat unrealistic assumption on the web) or perhaps more appropriately, one would have to use timestamps and versions. With timestamps, the non-monotone computation is turned into a monotone one over the versioned documents, and a formal semantics in the style of the one presented in Section 3 for the cumulative case may be defined.

We will not attempt to define a formal semantics for the non-monotone case. This is left for future research. Another important issue that we will little address in this paper is query evaluation. In our formal study, we will define the semantics of the document by looking at its fully materialized version. Therefore, the semantics of a query will be defined as the answer to the query on the fully materialized document where all function calls have been activated. Clearly, there is no need in practice to materialize everything. One only needs to materialize what is needed for the query or what is required by the attributes of function calls (such as *immediate*) appearing in the document. Not surprisingly, the optimization techniques needed to avoid full materialization lead to known techniques from a number of areas, most notably: (i) Query evaluation for mediator systems; (ii) warehouse maintenance for controlling the freshness of data; (iii) query evaluation in datalog programs [6, 31]; (iv) delta based techniques for describing changes; and (v) more generally, distributed query evaluation. This will have to be achieved in a context of limited capabilities at some sources. Thus the optimization problem is rather complex in the AXML framework.

By introducing limitations, one may simplify the problem. For instance, query evaluation is critically more complex in the presence of recursion. Also, detecting termination is simpler in a non-recursive context. By recursion, we mean that a function involving a document d_1 activates another function that uses a document d_2 that in turn activates a function that uses d_1. (Recursion may be defined with other granularities such as site or element level.) Forbidding recursion clearly simplifies the problem. However, although many applications do not require recursion, some do as illustrated in the following example inspired by [28]. Suppose that two sites cooperate to support some common yellow pages. In our terminology, *yp1* (and similarly *yp2*) are defined and used as follows:

```
let sc yp1.com/get-yp1() be
 for $a in
  document("yp1.com/yp1.axml")/entry
return { $a }
```

```
<axml id="yp1">
  <entry name="e1" val="v1"/>
  <entry name="e2" val="v2"/>
   <sc>y1.com/get-yp2()</sc></axml>
```

Clearly, this leads to some form of recursion.

To conclude this section, a last important aspect that will not covered, namely exception management, is mentioned. On the web, services will often not behave as expected. A service may not respond or it may return data of an incorrect type. This is only partly taken care of in WSDL and there are works on the topic, e.g., IBM HTTPR. Thus, there is a need to introduce an exception handling mechanism in the language.

3 Formal Semantics

We first introduce a simple data model for AXML documents. We present a formal semantics, in the absence of updates, for cumulative continuous services and obtain results for it. We then consider non-continuous services. For space reasons, we omit the proofs of our results.

3.1 Data model

We assume the existence of some infinite alphabets: *val* of data values, *element* of element names, *attribute* of attribute names, *document* of document names, *site* of site names, *function* of function names, *node* of node identifiers, *var* of variable names. We assume that all these alphabets are subsets of *string*, hence their values can be used in documents when needed. We also use a distinct symbol $\top$ for describing inconsistent data. To simplify the presentation, we ignore DTDs and XML identifiers. These can be added easily except for fusion based on XML identifiers that requires some care. We omit this here.

An (AXML) *instance* consists of a number of sites. Each site contains AXML documents and some function definitions. We next define instances, then proceed to the definition of documents and functions.

Instances An *instance* $\mathcal{I}$ is defined over a set of sites. The domain of $\mathcal{I}$ is a set of site names. The *content* of a site s, denoted $\mathcal{I}(s)$ is a tuple $(\mathcal{D}, \mathcal{F}, \mathcal{W}, sem)$ where

1. $\mathcal{D}$ is a set of document names, and for each d in $\mathcal{D}$, $sem(d)$ is an AXML document;

2. $\mathcal{F}$ is a set of function names, and for each f in $\mathcal{F}$, $sem(f)$ is a function definition; and

3. $\mathcal{W}$ (called the working area) is a set of document names (distinct from those of $\mathcal{D}$), and for each w in $\mathcal{W}$, $sem(w)$ is an AXML document.

All the sets are assumed to be finite. Intuitively, each document d in the working area of a site s represents the computation of some service call received by s. In particular, the document contains two special attributes, *site* and *node*, that determine where the result of this computation should be sent. They respectively identify, the calling site and the particular document element where the call was invoked. (The exact structure of these documents will be formally given in the sequel).

Documents An (AXML) *document* consists of a tree t and a function σ that describes the service calls used in t. The tree t is an unordered labeled tree over node identifiers (or is the inconsistent document $\top$). The nodes labeled with an element name are called *element nodes*, those with an attribute name are *attribute nodes*, and those with a value are *value nodes*. The other nodes (called *sc-nodes* for service call nodes) are not labeled. The tree edges correspond to the *component-of* relationship among document items and the leaves describe the values of attributes/atomic elements. We assume that the tree obeys constraints, some inherited from XML:

(*) An element node is not allowed to have two attribute children with the same label. An attribute has exactly one child that is a value node. Value nodes are leaves. *Service call nodes are leaves.*

The *content* of sc-node c, denoted $\sigma(c)$, is given by a tuple $\langle s, f, p1, \ldots, p_n \rangle$ where:

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

Note that in the AXML syntax that we used in the previous section, the content is typically in-lined with the rest of the AXML document. So, when we speak informally of an AXML document, this includes the content of embedded service calls.

When σ is understood from the context, we will speak of AXML document t.

Query language Function definitions will be based on XQuery. We also use XPath in function calls. The two query languages we use are essentially the conjunctive fragments of XQuery/XPath. They are obtained by eliminating from the two languages all non-monotone features such as negation of clauses. One difference is that we allow in AXML-XQuery the use of AXML data inside the *RETURN* clause of queries. For space reasons, we do not detail here the full syntax. The semantics are defined as follows. When applied on an inconsistent document, a query simply returns $\top$. Otherwise, the semantics is a natural adaptation of the standard semantics of XQuery/XPath:

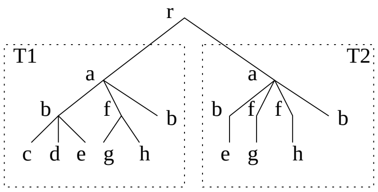
1. When evaluating path expressions, the service calls are simply ignored, and the path expression are matched against the "bare" XML skeleton of the data.
2. XQuery functions (e.g. $text()$) are applicable only to elements/variables whose value is a "standard" XML subtree (i.e. does not contain service calls or $\top$), and similarly for query predicates, (e.g. $=$, $\neq$) who otherwise evaluate to FALSE.
3. The *RETURN* clause of queries is interpreted in the standard way, using the variables instantiations to construct AXML documents.

Let $v_1, ..., v_n$ be variables. A *parameterized query* $Q(v_1, \ldots, v_n)$ is a query that may refer to the (variables) documents $v_1, \ldots, v_n$.

Function definition The *definition* of a function in a particular site s is a tuple of the form $\langle f, v_1, \ldots, v_n, Q(v_1, \ldots, v_n) \rangle$, where $f \in$ *function* is the function name, n the arity, $v_1, \ldots, v_n \in$ *var* are the parameters, and $Q(v_1, \ldots, v_n)$ is a parameterized query. The query Q may refer to documents in s and in no other site.

In practice, one may want to consider documents that are visible from other sites. Also, one may want to distinguish between *external* functions that are visible from all sites and *internal* functions. To simplify the presentation, these aspects will be ignored in the sequel.

Reduced documents Since we take here a monotone semantics, one subtree of an element may provide more information than another subtree. For instance, in the following tree, subtree t1 contains strictly more information than t2.



We will focus on normalized, reduced, documents with no redundancy. To define this formally, we first need to explain what it means for one document subtree to *subsume* another.

Definition 3.1 *Let t, t' be two (subtrees of) AXML documents. We say that t* subsumes *t', denoted $t \supseteq t'$, if $t = \top$, or if the following conditions hold:*

- *$root(t), root(t')$ have the same label or they both are sc-nodes.*
- *for every child subtree c' of $root(t')$, there exists some child subtree c of $root(t)$ s.t. $c \supseteq c'$.*
- *if t, t' are sc-nodes with respective service calls s, s', s* subsumes *s', denoted $s \supseteq s'$, i.e., they have identical site and function components, and for each parameter p_i, respectively p'_i, either they are both AXML trees with $p_i \supseteq p'_i$; or they are identical path expressions*[2].

When both $t \supseteq t'$ and $t' \supseteq t$ hold, we say that t and t' are equivalent*, and denote it by $t \equiv t'$.*

The query languages we use are the conjunctive fragments of XQuery/XPath. The main property of these fragments, that we use in the sequel, is monotonicity w.r.t to the $\supseteq$ relationship. This is formally captured by the following proposition.

Proposition 3.2 *Consider a parameterized conjunctive XQuery $Q(v_1, ..., v_n)$ and a conjunctive XPath query Q'. Let $\{t_i\}, \{t'_i\}$ be two sets of XML documents such that for each i, $t_i \supseteq t'_i$. Then $Q(t_1, ..., t_n) \supseteq Q(t'_1, ..., t'_n)$ and $Q'(t_1) \supseteq Q'(t_1)$*

We are now ready to define reduced documents.

Definition 3.3 *A document t is* reduced *if all the documents appearing in its service calls parameters are reduced, and it contains no node v with two children c, c' s.t. $c \supseteq c'$.*
A document is a reduced version *of document t if it is reduced and equivalent to t.*

The following proposition gathers some useful properties of subsumption and reduced documents.

Proposition 3.4 *(1) The relation $\supseteq$ is reflexive and transitive. (2) The relation $\equiv$ is symmetric, reflexive and transitive. (3) The reduced version of an AXML document is unique and can easily be computed.*

We will thus assume in the sequel that all our AXML documents are reduced. The notions of subsumption and reduction naturally extend to instances. An instance $\mathcal{I}$ is *reduced* if each document in each site is reduced. When the working areas in all sites are empty, the definition of subsumption for sites is straightforward: $\mathcal{I} \supseteq \mathcal{I}'$ if for each site s, $\mathcal{I}(s) = (\mathcal{D}, \mathcal{F}, \emptyset, sem)$, $\mathcal{I}' = (\mathcal{D}, \mathcal{F}, \emptyset, sem')$ and for every document d in $\mathcal{D}$, $sem(d) \supseteq sem'(d)$. The definition is slightly more complex for instances with non-empty working areas, and needs to account also for the subsumption and deletion of redundant working area documents. It is omitted here for space reasons.

[2]More generally, one could consider a more refined subsumption based on containment of path expressions. This is not needed here.

3.2 Non-deterministic Inflationary Fixpoint Semantics

We are now ready to define a non-deterministic inflationary fixpoint semantics for our AXML documents. We assume that data is not updated, that services are *continuous* and *cumulative* and that the mode is *immediate*. The non-deterministic fixpoint is used here to capture the essence of the *asynchronous* nature of the computation. The inflationary fixpoint we use is motivated by the fact that everything is monotone is this setting.

Intuitively, the semantics is defined as follows. The state of each site is a collection of trees. We activate service calls to derive more information about these trees. An activation of a service call spans a computation of a (possibly external) site. More precisely, the activation in Site A of a particular service call to Site B involves (1) (possibly) the evaluation at Site A of the XPath expressions in the call attributes, (2) the activation of concrete calls on Site A for each of the possible instantiations, and receiving the calls in Site B, (3) computing in Site B the corresponding answers and (4) sending back the answers to Site A, 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.

Note that, for continuous services, both Steps 1 and 3 above may need to be performed several times for a given service call: As time passes and the document gets richer, more path instantiations may be found at Step 1, so new concrete calls will need to be activated. Similarly, the result of a given instantiated query in Step 3 may get richer and these new results need to be sent to the calling site. This is captured by the following iterative nondeterministic computation.

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

Step 1: Instantiate the parameters of a call: For some sc-node c in $\mathcal{D}$ or $\mathcal{W}$, where some XPath expressions occur in $\sigma(c) = \langle s, f, p_1, \ldots, p_n \rangle$. The XPaths are evaluated, and for each instantiation a corresponding concrete service call becomes a sibling of c.

Step 2: Send/Receive an external call: For some sc-node c in $\mathcal{D}$ or $\mathcal{W}$ of site s without XPath attributes in $\sigma(c)$, the call to a remote site is activated. Formally, this consists in adding to the Working Area of the receiving site s', a new document with *site* attribute s, *node* attribute c, and an sc-node child r with $\sigma(r) = \sigma(c)$.

Step 3: Evaluate a local call: For some sc-node c in $\mathcal{D}$ or $\mathcal{W}$ without XPath attributes in $\sigma(c)$, and using some local function, evaluate the function query using the given parameters. The result, a forest, is inserted under the parent node of c.

Step 4: Return/Receive result of a call: For some document d of a Working Area with a *site* attribute s and a *node* attribute c, the children of $root(d)$, (not including the *site* and *node* attributes) are sent to site s and inserted there under the parent node of c.

Observe that this formal definition ignores a number of practical issues. First, we "packaged" *send* and *receive* in one operation, and similarly, the *return* and *receive* result. In practice, one site decides to activate a call (similarly for a result), sends it to the communication layer and this call will eventually reach the receiver. Intuitively, our send/receive operation captures the moment when the receiver receives the call. Also, nothing prevents (in this model) to do lots of redundant operations. For instance, the same call/answer may be sent several times. However, recall that we assume that all our instances are reduced. So, for example, sending the same call/answer many times will not affect the state. In practice, the implementation should clearly avoid doing unnecessary steps. Finally, note that one would introduce in practice a distinction between internal services and services that are visible from all sites. The semantics can easily be adapted so that AXML documents with only external services are exchanged between sites. We omit this here.

To guarantee a correct semantics, we need some *fairness* condition:

(†) Any operation that may happen eventually happens.

We say that $\mathcal{I} \rightarrow \mathcal{J}$ if $\mathcal{J}$ is obtained from $\mathcal{I}$ by one of the above transitions. We say that $\mathcal{I} \Rightarrow \mathcal{J}$ if instance $\mathcal{I}$ leads to $\mathcal{J}$ after some number of transitions. An initial instance $\mathcal{I}$ may be transformed in many different ways depending of the choice of the operations to perform. The document is enriched monotonously. A finite fixpoint is reached when no more operation can add something to the state. As shown in Example 2.2, there are cases when this never happens. We will see however that all computations lead to a unique state (which can be finite or infinite), i.e., the order in which the steps are executed is not important. As a first step, we show below that any data that can be derived in one particular sequence will also be eventually derived in any other sequence.

Lemma 3.1 Let $\mathcal{I}$ be an initial instance and suppose $\mathcal{I} \Rightarrow \mathcal{J}$ and $\mathcal{I} \Rightarrow \mathcal{K}$. Then, for each fair infinite sequence $\mathcal{J} \rightarrow \mathcal{J}_1 \rightarrow \mathcal{J}_2 \rightarrow \ldots$, there exists i such that $\mathcal{J}_i \supseteq \mathcal{K}$.

It should be noted that monotonicity is the only property required by the web services for the lemma to hold. So in practice, as long as they are monotone, arbitrary web services may be introduced in this framework (disregarding their actual implementation) and the result would still hold. The result suggests defining the semantics of an initial instance (Working Areas are empty) as: given a document name d, the semantic of d is the (possibly infinite) document obtained by removing the service calls from the instance obtained as the limit for d in an arbitrary (possibly infinite) computation. Based on the lemma, we have:

Corollary 3.2 The semantics of AXML document is well-defined.

Now, some AXML document may have a finite semantics and some do not. Using a reduction of the halting problem for Turing machines, one can show:

Theorem 3.3 Given an initial instance, one cannot decide whether it has a finite semantics.

3.3 Non-continuous services

Consider now non-continuous services, still assuming *cumulative* and *immediate* modes. We will assume here that all functions are non-continuous, but clearly, one may consider systems where some functions are continuous and some are not. In non-continuous services, we do things only once. In particular, we return the answer of a call only once. Note however that this does not mean that the work is finished once the answer is returned - the answer may contain new service calls that now need to be invoked. We will say that a service call has been *performed* if the call happened and it returned a result as well. We will say that a node is *complete* if all its children are known. Note that this means in particular that all the service calls that are children of this node should have been performed. On the other hand, some service calls in the subtree may still be pending.

For a given service call, each of Steps 1-4 is evaluated at most once. So, we have to be careful to have all the information needed at the time of the evaluation. More precisely, the possible instantiations for the path expressions in a call are computed once. For each instantiation, the function is activated only once. The corresponding query is evaluated exactly once, and a single answer is returned. For this to make sense, one such step is selected only when all operations that may affect this step have been performed. Let us analyze what these "*operations that may affect this step*" are. Not surprisingly, there are two facets to this issue that relate to the two query languages we use:

F1: XPath queries The instantiations of an XPath expression may change because some nodes have acquired more data because of the activation of some service calls.

F2: XQueries Similarly, the evaluation of a query also change.

Therefore, when we decide to instantiate an XPath expression or evaluate an XQuery, we must be sure that the data it bears on is stable. We will not detail this notion here. Intuitively, this means that the portion of the data that is used to evaluate the query will not change further because of the evaluation of some service call. More precisely, the query should not attempt to access descendants of incomplete nodes. When this is not the case, we will say that the XPath expression or the XQuery can be *fully evaluated*. Note that the result of the evaluation may still contain itself some unresolved service calls.

Another kind of problem comes from the fact that a path expression may change semantics if we modify its context. To see that, suppose that a service call c in a document t in Site 1 has the path expression "../../../a" for unique argument. Suppose that c appears in t as a child of the root, and that we want to send t to some Site 2, where is will be inserted deep in some local tree. The path expression has a different meaning in both sites. So we need to enforce the following:

F3: Context independence In subtrees that may be used in contexts different than their original one, always instantiate the path expressions that may be context dependent before using the subtree.

Again we will not detail here context independence. In short, it means that if we extract a piece of data (an AXML tree) from its original place in the tree, the path expressions occurring in this tree should evaluate strictly within the tree. Otherwise, we need to instantiate these path expressions before the tree is extracted.

We revisit the four steps of a computation to illustrate where these three facets come in the picture.

1. In Step 1, the instantiation of the path expressions in a call may only be performed when they can be fully evaluated (F1), and providing that all the selected subtrees are context independent (F3). When this is done, the call is marked as performed. Note however that it now has new siblings that are not performed.

2. In Step 2, a remote call can be performed at any time assuming (as before) that it includes no XPath attributes. Moreover, one must verify that context independence is not violated (F3).

3. In Step 3, the evaluation of a local service (a query evaluation), requires also that the query can be fully evaluated (F2). Also, one must verify that the all the variable instantiations used to construct the result, as well as the result itself, are context independent (F3). Once the query has been evaluated, the node is marked as performed.

4. For Step 4, consider a document in the Working Area. It starts with a single (instantiated) call as a child. We may decide to return the result only if this service calls is marked as performed.

Furthermore, one must verify that the returned result is context independent (F3). Both the *sc*-node that has activated the call (and receives the answer) and the document in the Working Area are marked as performed.

Using essentially a similar proof technique as for Theorem 3.1, we can show that the non-continuous semantics is also well defined and shares the same nice convergence properties. The undecidability theorem also still holds. Nevertheless, there are quite a lot of differences between continuous and non-continuous services. With the same document, less information will typically be derived with the non-continuous services. Indeed, some calls may never be activated, e.g., because some path expression can never be fully completed. In general, one cannot decide, given a document, whether its semantics with continuous or non-continuous services is the same. To conclude this section, we mention a particularly simple and easy case.

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 services without XPath expressions (neither in calls or in queries) *XPath-free* services. One can show, that for XPath-free layered services, (i) the semantics with continuous and non-continuous computations coincide, and (ii) the semantics is finite. In this context, it is simple to design evaluation strategies that detect termination.

We conclude this section with a remark on how to extend the semantics proposed in this section to handle the *lazy* mode, still in the *cumulative* case. Suppose a particular service call c is requested either because of a user request or because its evaluation is requested as immediate in the specification. Then c is marked as requested. All calls that may affect c (as analyzed in the previous section) and the calls that may affect those calls, and so on, are also marked as requested. The operation that is selected in Steps 1-4 is always a requested operation.

4 Architecture and implementation

First, we explain how the AXML model translates to the web environment of services, as it is currently being defined by the industry. We then briefly explain the development environment we used. Finally, we present the architecture of a system for providing AXML services and our ongoing implementation of an AXML server.

Towards industry standards We formally defined AXML documents in Section 3 as unordered XML documents containing particular function calls where the meaning of the functions is defined using queries. When it comes to implementation, one has to deal with *real* XML syntax and data, existing query languages for XML, and the particular protocols and standards needed to operate web services. This is considered next.

Service calls are expressed using an XML syntax, thereby leveraging existing XML-based tools such as parsers. The XML element describing a service call contains all the information needed to invoke the call, as dictated by the particular communication protocol or standard being used. In particular, we rely primarily on two standards, namely SOAP and WSDL, which operate at different levels. The Simple Object Access Protocol, SOAP [9], provides transport for method calls and answers on top of the common Internet protocols (HTTP, SMPT ...) and serialization conventions for useful programming data structures. The Web Services Description Language, WSDL [14], allows to describe with a higher level syntax, the operations exported by a service and hides the particular implementation details of the concrete underlying lower level protocols (e.g. SOAP).

The following is an example of the service call element for the *GetEmail* function of Section 2, describing the call at the SOAP level (the WSDL specification is similar):

```
<axml:sc
  endpointURL="http://xyz.com:80/soap/servlet/rpcrouter"
  nameSpaceURI="urn:xmethods-email"
  methodName="GetEmail" >
  <axml:params>
    <axml:param name="PesonName">
      <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 editors 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 namespace for them (the prefix *axml* in the example). As mentioned earlier, our data model ignores the order of elements in the document tree. Thus the order in which the data elements returned by the call are inserted into the parent element, is not guaranteed.

We use the WSDL standard to publish the interfaces of the service functions that are provided by a site. WSDL requires type information for input and output parameters of web services.

Development environment Java is used as the programming language because of its portability and of the availability of tools for XML processing. We also use the XOQL query processor [3]. In general, service functions may be defined using any XML query language or implemented using arbitrary programming languages. We chose XOQL because at the time we started this implementation, no XQuery processor was available to us. Also, we needed to adapt the query processor to the specific needs of AXML and a Java-based processor for XOQL had been developed in our group. The source code of this processor is freely available on the web. XOQL implements an algebra similar to the one of XQuery and is built on top of the Document Object Model (DOM). Although they are differences with XQuery, these are mostly syntactic.

The other third party modules that we rely on are all distributed under an open-source license. They all come from Apache [4]: (1) the Xerces Java XML parser, which provides namespace-aware parsing of XML documents; (2) the Tomcat servlet engine, (3) the Axis SOAP engine[3].

General architecture Before presenting the architecture of our AXML servers, we outline the main considerations that guided its design.

The AXML paradigm has a wide range of potential applications in a variety of contexts from PDA to large data warehouse. For instance, an AXML document may be used by a person to

[3]Although in early development stage, it provides good performance and great flexibility through its architecture based on chainable handlers.

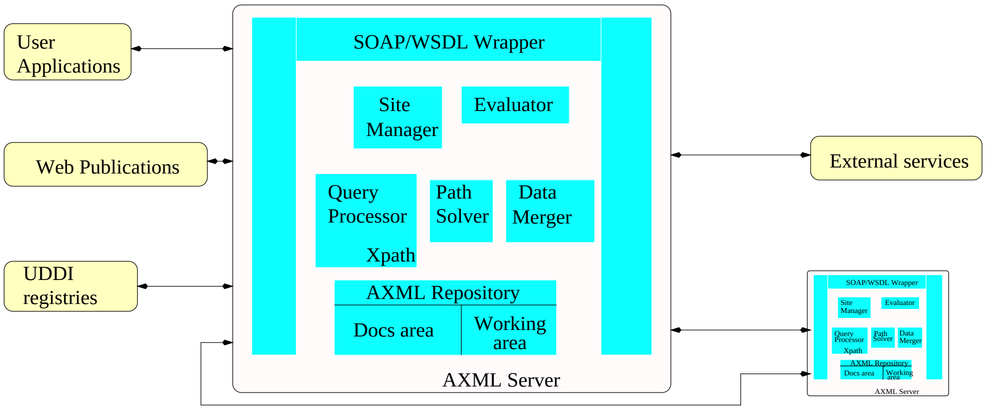


Figure 1: A simplified architecture of an AXML server

retrieve and update off-line in his/her PDA some critical data from a corporate web-site. At the other end of the spectrum, an AXML document may be used in a data warehouse to integrate considerable amounts of heterogeneous data from numerous sources scattered over the web. Therefore scalability in both directions is an essential requirement of the architecture. As a consequence, modularity is particularly important. A PDA may want to use only some modules, e.g., it surely needs to be able to call web services but may or may not need to be itself a server of AXML data. Also, one should be able to plug in different components for a given module depending on the context. For instance, AXML documents may be stored in a file system in a PDA and in industrial-strength databases for more heavy-duty services.

Following this, the architecture consists of the following seven modules. See Figure 1 for a simplified architecture of AXML servers. The seven modules are: (1) an AXML Repository, (2) A Query Processor (a processor of parameterized queries), (3) a Site Manager, (4) a SOAP/WSDL Wrapper, (5) a Path Solver, (6) a Data Merger, and (7) an Evaluator. Although this architecture is very general, we illustrate it with the particular implementation we are developing. We next briefly describe these components, and the role they play in the system.

The AXML Repository This module is in charge of managing the AXML documents of the site and of the working area. Because of XOQL, we rely on the DOM (level1) interface to interact with AXML documents. In the first prototype, the documents are loaded in main memory when needed, which is suitable for small to medium-sized repositories. One could also use any persistent DOM implementation such as X-Hive[36], or Tamino[26]. A comprehensive up-to-date review of existing products can be found online at [8].

The Query Processor This module, in charge of parameterized query processing, is an extension of the XOQL XML query processor. It manages free variables in query definitions. The variables must be bound to concrete parameters values before such a query is evaluated. By definition of query semantics (Section 3), the evaluation of a query ignores the service calls that may appear in the data (in particular, it never tries to resolve calls that it encounters on the way), and the path expressions are matched against the "bare" XML skeleton of the document. Implementation-wise, a great advantage of this is that our query processor can easily be replaced by a more efficient and/or standard-compliant one.

The Site Manager The site manager is a simple class that allows to add/modify/delete AXML documents and web service function definitions in the site. It is in charge of (WSDL) publication of the services of the site, i.e., the description of the functions and the adequate transport mechanism.

The SOAP/WSDL Wrapper This module is in charge of wrapping the XOQL queries as SOAP/WSDL web services, and sending/receiving requests and call results. It relies on the Apache Axis engine [5] and supports four handlers:

- SendCall: Forms a SOAP method call out of the given XML service call element, and sends it to the target site.
- ReceiveCall: Extracts parameter values from the received web service call and puts them, (together with the name of the of requested function and the return address), in the working area.
- SendResponse: Forms a SOAP response and sends it to the caller, and, if so directed, deletes the original request from the working area.
- ReceiveResponse: Extracts the call result and the identity of the origin node from the received response, and calls the *Data Merger* module (described below) to integrate it in the right place in the document.

The SendCall and ReceiveResponse handlers interact with web services defined in other sites, while ReceiveCall and SendResponse deal with answering calls to the server's services, coming from external sites.

The Path Solver By adapting the XPath capabilities of the XOQL query engine, this module resolves the XPath parameters of a given service call, while, if requested, testing some criteria for the nodes along the paths (e.g. completion of service calls, for non-continuous queries) and the selected subtrees (e.g. context independence). It is also in charge of adding to the document a concrete instantiated service call for each possible instantiation of the parameters. So this module is really central.

The Data Merger Given a node in the document, and another document tree, the module is in charge of merging the latter under the given node. The current version does not support id-based element merging. This will be added in the future.

The Evaluator The main role of this module is to select the service calls to be activated at each time. This selection is based on the particular evaluation strategy that has been chosen. The current version supports a single one strategy: immediate, continuous and cumulative evaluation. An important component of this module is the scheduler that records the validity times of every call, knows when the data is out-of-date and can thus triggers calls to refresh it.

To conclude this section, note that SOAP supports only very simple function calls. A function is called, returns an answer and terminates. To handle continuous services, we built a layer on top of SOAP [17]. The caller of a continuous service needs to use a web server (a proxy). The Caller web server calls the Service web server. Then each answer from Service (web server) is sent to Caller (web server). This is because typically the service is in a push mode, so messages are activated from the service side and not from the client side (as in a non-continuous web service).

5 Conclusion

The AXML language described in this paper allows to turn service calls embedded in XML documents into a powerful tool for data integration. This includes in particular (1) giving semantics to web services by means of parameterized queries on AXML documents, (2) supporting various integration scenarios like mediation and data warehousing, (3) utilizing (possibly infinite) stream services, and (4) distributing computations over the net via the exchange of AXML documents.

A formal foundation based on non-deterministic, inflationary fixpoint semantics was presented for the monotone case where service calls add new information but do not invalidate existing one. Non-monotone services introduce consistency problems and more research is needed to address them.

We implemented a first prototype. However, a lot of work is needed to develop efficient evaluation strategies and 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, and 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. Int. Conf. on Very Large Data Bases*, 1999.

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

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

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

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

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

[7] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *Tenth International World Wide Web Conference*, Hong Kong, China, May 2001.

[8] R. Bourret. XML Database Products. `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`.

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note, May 2000. `http://www.w3.org/TR/SOAP`.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000. `http://www.w3.org/TR/REC-xml`.

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

[12] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. W3C working draft, June 2001. `http://www.w3.org/TR/xquery`.

[13] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD*, pages 379–390, 2000.

[14] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Definition Language (WSDL) version 1.1. W3C Note, March 2001. `http://www.w3.org/TR/wsdl`.

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

[16] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath`.

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

[18] The Google Search Engine. `http://www.google.com`.

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

[20] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition.* Prentice-Hall, 1999.

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

[22] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 413–424, 1996.

[23] Jim Powell and Taylor Maxwell. Integrating Office XP Smart Tags with the Microsoft .NET Platform. `http://msdn.microsoft.com`.

[24] R. Preotiuc-Pietro, J. Pereira, F. Llirbat, F. Fabret, K. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proc. VLDB*, Cairo, Egypt, 2000.

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

[26] The Tamino XML Database. `http://www.softwareag.com/tamino/`.

[27] The Object Management Group. Common Object Request Broker Architecture (CORBA). `http://www.corba.org`.

[28] J. Trevor and D. Suciu. Dynamically Distributed Query Evaluation. In *Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 413–424, 2001.

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

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

[31] L. Vieille. Recursive axioms in deductive databases: The Query/Subquery approach. In *Proc. First Intl. Conf. on Expert Database Systems*, pages 179–193, 1986.

[32] 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.

[33] J. Widom. Research problems in data warehousing. In *4th Int'l Conference on Information and Knowledge Management*, 1995.

[34] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[35] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, DC, May 1993.

[36] X-hive Home Page. `http://www.x-hive.com/`.

[37] Xml-rpc Home Page. `http://www.xmlrpc.org/`.

[38] Xyleme Home Page. `http://www.xyleme.com`.