Towards a Flexible Model for Data and Web Services Integration

Serge Abiteboul, Omar Benjelloun, and Tova Milo

I.N.R.I.A., 78153 Le Chesnay, France
firstName.lastName@inria.fr

Abstract. We present a data-oriented approach to web services integration. To this end, we introduce a new framework based on a model that embeds service calls within semistructured data. The framework captures various integration scenarios including mediation and warehousing, and provides fusion based on object identifiers. Moreover, by allowing service call parameters and responses to contain calls to other services, the framework enables distributed computation over the web. This report describes our ongoing work on the subject, mainly through examples.

1 Introduction

There have been numerous works on data integration, see e.g., [9, 8]. We propose a new framework that allows to integrate data and *services*. We use a language based on XML documents, where some elements represent data and some represent service calls. The service calls are expressed as queries, more precisely as Datalog-style rules. The resulting documents (*Active XML* – **AXML** in short) will be described by examples. The syntax we use in the examples is invented for the occasion and borrows from many languages, most notably MSL [11] and ActiveViews [1].

The motivation for this work lies in recent developments related to the web [15]. The XML standard [4] allows exchanging semistructured data on the web. More recent proposals, namely SOAP [3] and WSDL [6], allow invoking web services and describing their interface. We are mostly concerned here with services that provide data. An example of such a service is a bibliography server that one may call, for instance, with an author's name, and that returns the list of papers published by the author. Another example that we will use in this paper is a service that provides temperatures for various cities. We are concerned here with XML documents that may include calls to such services.

An essential question in this framework is deciding when to call a particular service. For instance, the system may choose to do so when the information is needed, or periodically. The language allows to control the firing of these calls. Note that in mediator systems, the data is obtained only when needed, whereas in warehouses, data is loaded beforehand. In our framework, both approaches may be combined in a very flexible manner.

In general, services are rather simple: they receive XML data as input and they return XML data. Our model extends this scenario in two directions:

Subscriptions: New services are emerging that entail a possibly permanent relationship between the caller and the service, called a subscription [12, 5, 10]. Our framework supports such services. As we will see, subscriptions may be used in a variety of ways. First, they allow to include in an AXML document the messages of a particular query subscription in a cumulative manner. They also allow to propagate changes from external sources to the AXML document in a non-cumulative manner. The latter feature is in the style of the ActiveViews language. The request to be notified when a particular data item changes, or to receive each new value, is seen here as a subscription to a particular service.

Exchanging AXML: In standard services, only data is exchanged. In contrast, we allow the exchange of AXML documents that may contain calls to services. This is in the spirit of [13] where the answer to a query is allowed to include both extensional and intensional information. In our framework, it is not only the results of the calls that may contain service calls (intensional information) as in [13], but also their arguments.

We are currently working on a formal model and an implementation. The underlying structure is roughly XML without order (labeled trees). The computation model is an extension of Datalog, see [14, 2]. Datalog is first extended in a standard manner with trees and object identifiers (Skolem functions), as in MSL. Service features are implemented by adding the possibility to derive more rules. Finally, we introduce update features based on inflationary or non-inflationary fixpoint logic. The formalization is on-going and not presented here, and similarly for the implementation.

In the first sections (Section 2 to Section 7), we set up the machinery needed for data and services integration. The issue of distribution is addressed in Section 8.

2 Data Integration

Data integration has been extensively studied. Typically, the specification of such integration is based on a query language. For instance, we may want to state that the information we have about cities comes from two source relations:

```
s1{capital{name{"Paris"},country{"France"},
           climate{"moderate"}}}; ...;
s2{capital{name{"Mexico"},country{"Mexico"},
           climate{"cold"}}}; ...;
s1{ capital{name{$X},country{$Y},climate{$Z}} :-
    s2{capital{name{$X},country{$Y},climate{$Z}}} };
s2{ capital{name{$X},country{$Y},climate{$Z}} :-
    s1{capital{name{$X},country{$Y},climate{$Z}}} };
```

We are dealing here with two XML documents, `s1` and `s2`. Each is defined partially extensionally and partially by a query (in an MSL style language) attached to its root. The intuition is that each one of them knows all the information about capitals that the other has. More generally, in AXML documents, data and queries can be placed at any node of the trees. Observe that the language allows recursion.

3 Identifier-based Integration

The previous example illustrates "vertical/union" integration. Clearly, one can also define "horizontal/join" integration. More interestingly, one can perform "fusion" based on identifiers. To do that, we use semantic identifiers (Skolem functions from a technical viewpoint). For instance, we can incorporate to `s2` data coming from another source using fusion based on the `name` attribute:

```
s2{ [$X]:capital{name{$X},country{$Y}} :-
      s2{capital{name{$X},country{$Y}}} };
s2{ [$X]:capital{continent{$W}} :-
      t2{capital{name{$X},continent{$W}}} };
```

The first rule simply appends Skolem names to existing pieces of data. The symbol "[$X]" specifies that the name of the capital should be taken as identifier for the particular element. The second rule merges data from a different source.

Because the model is slightly more complex than the relational model, we immediately face consistency issues. An inconsistency may occur, for instance, if we assign the same identifier to two nodes in the tree with distinct labels or if we assign two values to the same attribute of a particular element.

4 Service Integration

Let us come now to a more novel aspect, the integration of services. We can think of Web services in many different ways. A service may be viewed as a method call (a SOAP or Corba call). We give it arguments and it returns data. We can also think of it as a form on the Web that we fill (with arguments) and that provides a new Web page (returned result). Finally, we can think of it, the approach we follow, as a query with parameters. Indeed, we can reconsider the second rule in the example of Section 3 based on services (queries):

```
s2{ Q() };
let Q() be { [$X]:capital{name{$X},continent{$W}} :-
                t2{capital{name{$X},continent{$W}}} };
```

Document `t2` may reside on some site `s` and `Q` may be defined there. Document `s2` may be on a different site that only needs to know that `Q` exists and where it is defined. We will come back to this aspect in more depth in Section 8.

Remark 1. When all the needed data is local, it is strictly equivalent to use the definition given in Section 3, that doesn't explicitly name the query, or this last one, where the query is named. When we will move to distributed data, the need for query definitions will become obvious.

To see a service with parameters, suppose that the French Press Agency provides a Web service called `MeteoAFP` that, given a city name, returns the current temperature of the city. This service could be defined with a query as follows:

```
let MeteoAFP($X) be { $Y :- meteo{city{$X},temp{$Y}} };
```

The core of this definition is a query to some (virtual) relation `meteo` that is supported by the site of AFP and is invisible outside of that site. To use this service, it suffices to know its signature. Using this service, `s1` may also provide temperatures using statements such as:

```
s1{capital{name{"Paris"},country{"France"},climate{"moderate"},
        temp{MeteoAFP("Paris")}}};
```

or similarly:

```
s1{capital{name{"Paris"},country{"France"},climate{"moderate"},
        temp{MeteoAFP(../name)}}};
```

In the last rule, the argument of the service call is an XPath [7] expression that is evaluated at the `temp` element. The `".."` brings to its parent, the `capital` element; and `"/name"` evaluates to `"Paris"`. Observe that calling `MeteoAFP` with argument `"../name"` does not make sense – `"../name"` does not have any meaning outside the current document. Thus, we have to evaluate (if possible) `"../name"` before calling the service. Note also that `"../name"` does not have to be simple data. It may itself contain service calls. This is considered in the next section.

The granularity of the definition of `temperature` does not have to be limited to a single city. For instance, suppose that AFP provides temperatures for European cities and CNN for cities in the American continent. We may use the following specification in `s2`:

```
s2{ Q1() ; Q2() };
let Q1() be { [$X]:capital{temp{MeteoAFP(../name)}} :-
    s2{[$X]:capital{continent{"Europe"}}} };
let Q2() be { [$X]:capital{temp{CNNMeteo(../name)}} :-
    s2{[$X]:capital{continent{"America"}}} };
```

This states that each European capital has a temperature that is obtained by calling `MeteoAFP`, passing it as argument the name of the capital; and similarly for American capitals.

5 Using calls in parameters and query results

The services we considered so far get data (XML documents) as parameters and also return XML documents. One might consider more sophisticated services, capable of exchanging both extensional and intensional information, i.e. trees containing both data and service calls. To see an example, suppose we do not know the capital of Spain but we know that Yahoo (YC service) knows it. Suppose further that the AFP service does not know the temperature of Madrid but knows that the Spanish weather forecast system, SMS, does. Two possible sequences of evaluation that may happen if we ask for the temperature of the Spanish capital are:

Scenario 1	Scenario 2
YC("Spain") → "Madrid" MeteoAFP("Madrid") → SMS("Madrid") SMS("Madrid") → "25c"	MeteoAFP(YC("Spain")) → "25"

In the first scenario, we directly call YC, whereas in the second one `MeteoAFP` does so. Observe that in the first scenario `MeteoAFP` returns an intensional answer. This is in the spirit of [13]. In the second case, we send to `MeteoAFP` AXML data containing method calls. Clearly, we need to take into consideration service capabilities. It may be the case that `MeteoAFP` only accepts "pure" XML data as parameters, which would make Scenario 2 unfeasible.

Note that we are now using AXML documents containing service calls both as parameters and in results. If we assume that the services are distributed, this can be viewed as shipping programs over the web, so we have moved to distributed computation over the web.

6 Validity time

Each piece of information we obtain comes equipped with attributes *source*, *acquisition time* and *validity* (i.e., the time until which it remains valid). The validity attribute may either be defined by the source, or by default, or explicitly by the rules. This last kind of definition is illustrated next:

```
s1{capital{name{"Paris"},country{"France"},climate{"moderate"},
    temp{MeteoAFP("Paris")[@valid{"2 hours"},@mode{"lazy"}]}}};
```

The attribute "@valid" is attached to the value returned by the service call, here a PCDATA. (Note that it is not possible, in the standard XML model, to attach an attribute to such a node. We extended the model to allow it.) The lazy mode indicates that the system waits until someone requests the temperature before calling AFP to obtain it. The validity of this information is 2 hours, that is if someone requests the temperature 2 hours later, a new call is issued. In a non-lazy mode, the service would be called as soon as the temperature data becomes stale. Note that the model allows to maintain information in many different styles:

– mediator style: by simply fixing the validity time to 0;

– mediator style with caching: non 0 validity time and lazy mode;
– warehouse mode with pulling information: non 0 validity time and non lazy mode.

A last feature (stream of results) considered in the next section will allow us to have a "clean" warehousing style where sources directly push their updates to applications.

7 Streams of Results

We first consider a "cumulative" use of a stream of answers that typically captures query subscriptions. Then, we consider a "non-cumulative" use that will lead to the "clean" warehousing mentioned in the previous section.

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 though where we have to consider a possibly infinite stream of answers. For instance, with the Xyleme [16] subscription system [10], one may ask to be notified each time the crawler discovers a new document containing some specific keywords. Such a service, called XyWatch, can be used in our setting, and indeed this is one of the motivations of the present work. The following example illustrates the use of such services:

```
s3{keywords{"database XML"}};
s3{keywords{"sculpture Rodin"}};
s3{ [w($X)]:watch{keywords{$X},res{XyWatch($X)}} :-
     s3{keywords{$X}} };
```

The last rule starts a new subscription for each sequence of keywords. The (infinite) stream of results for this subscription is inserted in a XyWatch object. After a few subscriptions, the general structure of the data may look like:

```
s3{keywords{"sculpture Rodin"},
   keywords{"database XML"},
   watch{keywords{"sculpture Rodin"},res{subs{...},...}}},
   watchch{keywords{"database XML"},res{subs{...},...}}}.
```

In this manner, we have open connections and are able to record flows of data coming from services.

We saw how an external source may send a flow of data that we were simply cumulating at the receiving node. This *cumulative* approach simplifies the underlying foundation: we remain in a Datalog framework with some inflationary semantics. Now suppose that a weather forecast source can send the temperature of a capital each time it updates that temperature. This allows us to have fresh information without having to perform new service calls. From an implementation viewpoint, this is exactly like a query subscription. We receive a flow of answers. The only difference is in our interpretation of answers. Each new temperature invalidates the previous one. So, for instance, suppose that AFP supports such a service under the name `OnlineMeteoAFP`. We may request to be constantly updated on changes of the temperature in Paris:

```
s1{capital{name{"Paris"},country{"France"},climate{"moderate"},
        temp{OnlineMeteoAFP("Paris")[@valid{last}]}}};
```

The value of the `@valid` attribute specifies that only the last received temperature should be considered as valid. Observe a difference with the management of temperature using standard queries. Here the temperature data is pushed to us by the service. We need to call the service only once to subscribe to it.

The definition of `OnlineMeteoAFP` resembles that of `MeteoAFP`:

```
  let OnlineMeteoAFP($X) be continuous
      { $Y :- meteo{city{$X},temp{$Y}} };
```

The keyword `continuous` indicates that the service is a continuous query in the style of, e.g., [5]. The user of such a service needs to understand the precise semantics: Is a new value sent each time the result of the query changes ? Is this result sent periodically, e.g. daily ?

To conclude this section, observe that by removing the mode "last" in the example, we can specify that the incoming data should be "versioned". This is because the default semantics is inflationary, and each call result has attributes that provide time stamps (see Section 6). The impossibility to attach attributes to atomic values (PCDATA) in the standard XML model can be dealt with by introducing a particular `pcdata` element and attaching attributes to it. For instance, the value of the `temperature` of Paris may look like:

```
temp{pcdata{"25",@date{"25/01/2001,10:44"},@source{...},...},
     pcdata{"27",@date{"25/01/2001,12:31"},@source{...},...}}
```

8 Distribution

Although we have mentioned distribution, our examples so far assumed that all AXML documents were on the same server. Now let us consider three sites: `location1`, `location2` and `AFP`. Each site publishes the signature of the services it provides. Say we have:

```
AFP: MeteoAFP(X)
location1: Q1()
location2: Q2()
```

Each document or query at a particular site may refer to the documents of the site, its own queries and the queries published by other sites. For instance, a document at `location1` may refer to document `s1` and query `Q2`. This is illustrated in the following example:

```
location1
  s1{capital{name{"Paris"},country{"France"},
             climate{"moderate"}}}; ...;
  s1{location2/Q2()}
  let Q1() be
      { capital{name{$X},country{$Y},climate{$Z}} :-
          s1{capital{name{$X},country{$Y},climate{$Z}}} };
location2
  s2{capital{name{"Mexico"},country{"Mexico"},
             climate{"cold"}}}; ...;
  s2{location1/Q1()}
  s2{localQ()}
  let private localQ() be
      { [$X]:capital{AFP/MeteoAFP($N)} :-
         s2{[$X]:capital{name{$N}}}} };
  let Q2() be
      { capital{name{$X},country{$Y},climate{$Z}} :-
          s2{capital{name{$X},country{$Y},climate{$Z}}} };
AFP
  meteo{city{"Paris"},temp{"25"};...};
  let MeteoAFP($X) be { $Y :- meteo{city{$X},temp{$Y}} };
```

Here, locations are just symbolic names that encapsulate a number of informations such as the HTTP address, the port and service name, etc., as described in the WSDL specification [6].

When a (non-local) service is called, the parameters it receives should all have a meaning independently of the calling site. Similarly, one site should return an AXML document that may not be fully extensional, but should be understandable by the caller. A service call leads to starting a computation at the service site. This computation continues until it decides to return some result. In the case of "normal" queries, this particular computation at the service site finishes. In the case of a continuous query, it continues.

AXML documents may also be viewed as web agents. They may start new agents via service calls. One could also consider shipping XML documents around in a mobile computing style. The only constraint is that units that are transfered should be meaningful.

9 Conclusion

The work presented here is on-going. We have developed a prototype that accepts XML documents with included SOAP calls. We have also developed a SOAP implementation of query subscriptions. We intend to complete the formalization of this integration model and to implement it.

References

[1] S. Abiteboul, B. Amann, S. Cluet, E. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. Online at http://www-rocq.inria.fr/verso/ACTIVEVIEWS/paper/av.pdf, 1999.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note, May 2000. http://www.w3.org/TR/SOAP.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000. http://www.w3.org/TR/REC-xml.

[5] 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.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Definition Language (WSDL) version 1.1. W3C Note, March 2001. http://www.w3.org/TR/wsdl.

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

[8] H. Garcia-Molina et al. The TSIMMIS approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8:117–132, 1997.

[9] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, pages 251–262, 1996.

[10] B. Nguyen, S. Abiteboul, G. Cobena, and L. Mignet. Query subscription in an XML webhouse. In *Workshop on Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[11] Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Object Fusion in Mediator Systems. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, pages 413–424, 1996.

[12] Radu Preotiuc-Pietro, João Pereira, François Llirbat, Françoise Fabret, Kenneth Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In *Proc. VLDB*, Cairo, Egypt, 2000.

[13] Jim Trevor and Dan 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.

[14] J. Ullman. *Principles of Database and Knowledge-Base Systems, volume 2*. Computer Science Press, 1989.

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

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