Positive Active XML

Serge Abiteboul
INRIA & Xyleme
Serge.Abiteboul@inria.fr

Omar Benjelloun
INRIA
Omar.Benjelloun@inria.fr

Tova Milo
INRIA & Tel-Aviv University
Tova.Milo@inria.fr

Abstract The increasing popularity of XML and Web services introduced a new generation of documents, called Active XML documents (AXML), where some of the data is given explicitly while other parts are given intensionally, by means of embedded calls to Web services. Web services in this context can exchange intensional information, using AXML documents as parameters and results.

The goal of this paper is to provide a formal foundation for this new generation of AXML documents and services, and to study fundamental issues they raise. We focus on Web services that are (1) monotone and (2) defined declaratively as conjunctive queries over AXML documents. We study the semantics of documents and queries, the confluence of computations, termination and lazy query evaluation.

1 Introduction

XML, a self-describing, semistructured data model, is becoming the standard for data exchange between applications over the Web. Complementarily, recent standards for *Web services* such as SOAP [21] and WSDL [24], normalize the way programs can be invoked over the Web. Together, XML and Web services are becoming the standard means of publishing and accessing valuable, dynamic, up-to-date sources of information. The increasing acceptance of these standards naturally lead to the introduction of a new generation of XML documents, where some of the data is given explicitly, while other parts are given only intensionally, by means of embedded calls to Web services [1, 19, 17]. We refer to such documents as *Active XML documents* (AXML documents for short) [11]. Web services in this context can exchange intensional information, by using AXML documents as parameters and results. We refer to services exchanging AXML data as *AXML Web services*.

Consider as a simple example an AXML document that describes the content of a jazz portal. This document contains some extensional information, e.g. a list of jazz CD's reviewed by the portal, as well as some intensional information, e.g. how other lists of jazz CD's and reviews may be obtained from other portals, via calls to some Web services. This intensional information may be materialized by invoking the services, and the list of references of the portal thereby enriched. The portal also provides an AXML Web service that, given some search criteria (e.g. the name of a musician), returns the desired information. The answer is an AXML document that may contain, besides extensional reviews, embedded calls to other portals to obtain more information and eventually obtain music.

The goal of this paper is to provide a formal foundation for AXML documents and services, and to study fundamental issues they raise. The main aspects that are considered are the following.

Confluence In general, in AXML, the state of the system may depend on the order of service call invocations. We focus here on AXML Web services, that enrich the documents in a *monotone* manner, and on *fair* sequences of call invocations, namely sequences where each call that may bring new data is eventually invoked. We demonstrate a form of confluence for this setting.

Recursion and termination A call to a service may activate a call to another service, and so on, possibly recursively. Also, a service may return as answer some data including new calls, which may in turn return more data including more calls, etc. Therefore, some computations may never terminate. We study termination detection and enforcement, for both documents and query evaluation.

Lazy evaluation When querying an AXML document, it may be unnecessary to invoke *all* the service calls and materialize the full documents to answer the query. One would like to focus on *relevant* calls only. This notion of relevance is formally defined in the sequel, and some fundamental results on lazy query evaluation are presented.

To study the above issues, we use a simple model where AXML documents are modeled as unordered labeled trees having two kinds of nodes, data nodes and function nodes (the latter represent calls to Web services). The semantics of documents is defined as the trees obtained at the limit of an arbitrary fair sequence of service invocations. The multiple invocations of services is meant to capture P2P data management based on streams of data (both in pull and push mode).

In this paper, we consider only monotone Web services. The documents containing the calls are monotonously enriched by the answers. This is in the style of peer-to-peer computations ala Kazaa [18] where data is incrementally collected by some peers (acting as servers) and sent to others (the clients). Non monotone computations are hard to en-

vision in such a setting: one can never assume that a fact is false since this fact may be stated in some parts of the network not investigated yet. We will ignore non monotone services in this paper.

When services are monotone, the semantics of a document is possibly infinite but unique (up to an equivalence relation), i.e. independent of the order of service invocations. A further analysis may be performed if the semantics of services is known. In particular, we consider a class of AXML documents and services, which we call *positive AXML*, where Web services are defined using a monotone query language that corresponds to the core tree-pattern fragment of XQuery. Because of the recursion between documents and services, we still obtain an important expressive power. In particular, a large class of Turing machine computations can be simulated.

We also highlight some good properties of a particular subclass of these queries, which we call *simple queries*, obtained by disallowing variables that range over subtrees of documents. Intuitively, such variables can be used to copy subtrees of arbitrary complexity. For instance, termination is decidable when AXML services are defined by simple queries (while it is undecidable in general). Furthermore, one can always construct a *finite graph representation* of the (possibly infinite) document semantics. This finite representation also facilitates lazy query evaluation. In particular, it allows to detect which service calls are relevant for query evaluation and which are not, a property that is also undecidable in general. The above problems (and related ones) are shown to be CO-NP hard for simple services, and EXPTIME algorithms to solve them are given. As these may be too expensive in practice, we also consider alternative PTIME heuristics.

AXML documents and services were originally introduced in [2]. A first version of an AXML system was presented in [1]. In this, and in follow-up works on AXML [3, 20], the model was only informally sketched. The system uses the full XML syntax, in particular, trees are ordered whereas we see them as unordered; and services are defined using a full fledged XQuery-like language while we use a more limited monotone language. It also has a number of complex features not discussed here, e.g. updates. The formalization presented here is new, and so are the results. Since XML and Web services are promised such a brilliant future, we believe it is very important to develop a formal foundation for AXML, so that this technology can be better understood and used. Clearly, the general problem is complex, and further work is needed. While our research originated from the AXML system, the results are more generally applicable to other systems supporting data with embedded service calls, e.g.[19, 17].

The paper is organized as follows. The monotone AXML data model where services are arbitrary monotone functions, is defined in Section 2. Positive AXML systems, where services are defined by queries in a particular language, are studied in Section 3. Lazy query evaluation is considered in Section 4. To simplify the presentation, the previous sections use a very simple query language. Extensions to this language are considered in Section 5. We review related work and conclude in Section 6.

2 Monotone Active XML

In this section, we formally present AXML documents and systems, i.e., those using *monotone* Web services. We first ignore service implementations, thus viewing them as black-boxes. Then, in the next section, we will consider a particular class of monotone services, called *positive* services, that are defined by conjunctive queries over AXML documents.

2.1 AXML documents

AXML documents are modeled as unordered labeled trees with *data* and *function* nodes. The function nodes correspond to service calls, and their children subtrees represent call parameters. We assume the existence of some disjoint domains: $\mathcal{D}$ of document names, $\mathcal{N}$ of nodes, $\mathcal{L}$ of labels, $\mathcal{F}$ of function names[1], and $\mathcal{V}$ of atomic values.

Definition 2.1 *An (unordered)* AXML document *(a* document *for short) is an expression* (T, λ)*, where* $T = (N, E)$ *is an unordered tree,* $N \subset \mathcal{N}$ *is a finite set of nodes,* $E \subset N \times N$ *are the edges, and* $\lambda : N \to \mathcal{L} \cup \mathcal{F} \cup \mathcal{V}$ *is a function over nodes, and such that (i) only leaf nodes may be assigned atomic values and (ii) the root is assigned a label or an atomic value.*

For a node n, $\lambda(n)$ is called its *marking*. Nodes with a marking in $\mathcal{L} \cup \mathcal{V}$ are called *data nodes* while those with a marking in $\mathcal{F}$ are called *function nodes*. The children subtrees of a function node are the *call parameters*. We will use a compact syntactic representation of trees. In the examples, we represent labels with strings, function names with bold strings and atomic values with quoted strings. A sample document is as follows:

```
directory{cd{title{"L'amour"},
             singer{"Carla Bruni"},
             rating{"***"}},
          cd{title{"Body and Soul"},
             singer{"Billie Holiday"},
             GetRating{"Body and Soul"}},
          cd{title{"Where or When"},
             singer{"Peggy Lee"},
             rating{"*****"}},
          FreeMusicDB{type{"Jazz"}},
          GetMusicMoz{
             FindSingerOf{"Hotel California"}}}
```

[1] Intuitively, function nodes correspond to invocations of Web services. So, the real world analog of a function name involves notions such as service URL and operation name, that are needed to invoke the service.

This tree contains *cd* nodes with *title*, *singer*, and *rating* children. For some *cd*s, the rating is given explicitly, while for others it may be obtained by calling the ***GetRating*** function. The tree also contains a function node, ***FreeMusicDB*** returning more data on jazz, and a function node ***GetMusicMoz*** searching for more songs by the singer of "*Hotel Califonia*". Observe that call parameters may themselves contain function nodes. Here, the parameter of ***GetMusicMoz*** is a call to the ***FindSingerOf*** service, retrieves the singer name.

When a function is called, the parameter subtrees are passed to it. The return value, a forest of AXML documents, is then appended as sibling of the function node in the document. We define this process formally later on.

Reduced documents In this paper, we focus on monotone information and monotone functions, which motivates the following definition of reduced documents. Intuitively, consider a node n with two subtrees t, t' with t' containing strictly more information than t, then t is basically useless (from the viewpoint of the query languages considered here).

To formally define reduced documents, we use the auxiliary concept of tree *subsumption*.

Definition 2.2 *A document (T_1, λ_1) is* subsumed *by a document (T_2, λ_2), denoted $(T_1, \lambda_1) \subseteq (T_2, \lambda_2)$, if there exists a mapping h from the nodes of T_1 to those of T_2, mapping the root of T_1 to that of T_2, preserving the parent-child relationships among nodes and the marking of nodes (i.e., $\lambda_1(n) = \lambda_2(h(n))$ for each n).*

For two documents d_1, d_2, when both $d_1 \subseteq d_2$ and $d_2 \subseteq d_1$ hold, we say that d_1 and d_2 are equivalent*, and denote it by $d_1 \equiv d_2$. A document d is said to be* reduced *if there is no subtree* [2] *of d equivalent to d. A* reduced version *of a document d is a reduced document d' equivalent to d.*

For instance, the document $a\{b\{c,c\},b\{c,d,d\}\}$ is not reduced, since $b\{c,c\} \subseteq b\{c,d,d\}$. The tree $a\{b\{c,d\}\}$ is its reduced version. We can prove:

Proposition 2.1 *(1) Document subsumption is a transitive and reflexive relation. (2) Each document has a unique (up to node isomorphism) reduced version. (3) Subsumption can be tested in* PTIME. *(4) A reduced version of a document can be computed in* PTIME.

Proof: (Sketch) The transitivity and reflexivity follow immediately from the definitions. We show that the mapping h can be constructed in PTIME, by first building a simulation relation [16] between the trees, and then trimming it down to form the desired mapping. A reduced version of a tree can then be constructed by a bottom-up elimination of subsumed siblings. Its uniqueness follows from the transitivity of the subsumption relation and the minimality of the reduced tree. ■

[2] A *subtree* of a tree d is a tree whose nodes and edges are subsets of the nodes and edges of d.

The notions of document subsumption, equivalence, and reduction, extend naturally to *forests*, i.e. *sets* of documents. A forest φ is subsumed by a forest φ' if each tree in φ is subsumed by some tree in φ'. A forest φ is *reduced* if all its trees are reduced, and no tree is subsumed by another. *In the following, we identify each document (resp. forest) with its equivalence class, and use the reduced version as a representative for that class.*

Related to these, we also use the following notion. Given two documents d and d', with the same root label, one can verify that they have a *least upper bound* (w.r.t. $\subseteq$), up to equivalence. We denote it by $d \cup d'$. It can be obtained by constructing (and then reducing) a tree having the same root label as d and d', and having as children subtrees all the children subtrees of the roots of d and d'. Trees with distinct root labels are incomparable.

Remarks: Observe the following.

- Document subsumption ignores function semantics. For instance, even if for some functions $\boldsymbol{f}$ and $\boldsymbol{g}$, for any x, $\boldsymbol{f}(x) \subseteq \boldsymbol{g}(x)$, still the documents $a\{\boldsymbol{f}\{\text{“5”}\}\}$ and $a\{\boldsymbol{g}\{\text{“5”}\}\}$ are incomparable. If more information were available on services, one could define a more powerful notion of subsumption, by considering also function subsumption. For instance, if services are defined as queries, one could define it as query containment (introduced further). We will not do it here.
- One could also introduce node identifiers (ala IDref). Subsumption would then require the mapping to preserve node identifiers. This will also be ignored here.

2.2 Monotone AXML systems

A monotone AXML system consists of a set of AXML documents, plus the services used in these documents.

We first ignore the specification of Web services, and view them as "black-boxes". In this view, a *Web service s over* a set of document names $\{d_1, \ldots, d_n\}$ *using a set F of function names* is defined as follows. Given an assignment θ, mapping $d_1, ..., d_n$ to AXML documents, $s(\theta)$, returns a forest of AXML documents with only function names in F. We will see that services may use two reserved document names, $input$ and $context$, that represent respectively the call parameters and context (to be defined precisely in the sequel).

We focus here on *monotone services*. These are services s such that, for each θ, θ', if for every i, $\theta(d_i) \subseteq \theta'(d_i)$, then $s(\theta) \subseteq s(\theta')$.

Definition 2.3 *A* monotone AXML system, *(a* system *for short) is an expression (D, F, I) where*

- *D is a finite set of document names, not containing* input *and* context*;*
- *F is a finite set of function names;*

- *I is a mapping over $D \cup F$ such that for every $d \in D$, $I(d)$ is a document with only function names in F, and for every $\boldsymbol{f} \in F$, $I(\boldsymbol{f})$ is a monotone service over $D \cup \{input, context\}$ using the set F of function names.*

(To be precise, we assume that documents do not have nodes in common.)

For brevity, when D and F are understood from the context, we will simply use I to denote a system. The fact that $I(d) = t$ for some document name d and tree t, is denoted *d/ t*. An example of a system is I_1, containing *d/ a{b,c}* and *d'/ a{**f**{c}}*, and such that $I_1(\boldsymbol{f})$ is some monotone function.

To define the semantics of monotone AXML systems, we need first to define formally the semantics of service call invocations.

Service call invocation Consider a document d in a system I, and a function node v marked $\boldsymbol{f}$ in it. When the function is *invoked*, $I(\boldsymbol{f})$ is evaluated, and its result is appended to the document, as siblings of the node v.

To define this formally, we first need to give a meaning θ to the document names d_i used by the function, namely to $input, context$ and the names in D. The meaning of *input*, $\theta(input)$, is the tree with a root labeled $input$ and all the subtrees of v as children. The meaning of *context*, denoted $\theta(context)$, is the subtree rooted at the parent of v. The meaning of the names in D is the one given by I. Let the forest φ be the result of the function (i.e., $\varphi = I(\boldsymbol{f})(\theta)$). Then, the result of the *invocation* of v is obtained by appending φ to the tree, as siblings of v (assuming the nodes of φ are disjoint from those of I.)

For instance, consider the AXML music directory presented above and assume that the ***GetRating*** function in it is invoked. Here the meaning of *input* is the tree "*Body and Soul*". Assume that the function returns the tree *rating{"*****"}*. The document, after the function invocation has the following form.

```
directory{...
          cd{title{"Body and Soul"},
             singer{"Billie Holiday"},
             GetRating{"Body and Soul"}
             rating{"*****"}},
          ...}
```

Sequence of invocations The *semantics* of an AXML system is defined to be the set of (possibly infinite) documents obtained at the limit of an arbitrary *fair* sequence of service invocations, i.e., a sequence where any function invocation that may bring new data eventually takes place.

Definition 2.4 *For a system I, we say that $I \xrightarrow{v} I'$ if I' is obtained from I by the invocation of some function node v in I, and $I \not\equiv I'$. A (possibly infinite)* rewriting sequence *is a sequence $I \xrightarrow{v_1} I_1 \xrightarrow{v_2} I_2 \rightarrow \ldots \xrightarrow{v_n} I_n \ldots$. We say that I* rewrites *to I_n, denoted $I \xrightarrow{*} I_n$. We say that the system* terminates *at I_n if there is no function node v_{n+1} in I_n and no I_{n+1} s.t. $I_n \xrightarrow{v_{n+1}} I_{n+1}$. An infinite sequence is said to be* fair *if for any i and any function node $v \in I_i$ there exists $j > i$ s.t. at least one of the following conditions hold: (i) $I_j \xrightarrow{v} I_{j+1}$, or (ii) an invocation of v would not modify I_j.*

Observe that $I \xrightarrow{v} I'$ requires that $I \subseteq I'$ and $I \not\equiv I'$. So, the rewriting is strictly increasing, i.e., the documents are enriched by this rewriting.

To be able to formally define the semantics of monotone AXML systems, we just need the last notion of *infinite AXML document*. An infinite AXML document is one where the set of nodes is not restricted to be finite. The definition of an *infinite monotone AXML system* is obtained from that of a monotone AXML system by allowing documents to be infinite. (Note that we still assume that there are finitely many documents and functions.)

Definition 2.5 *Let I be a monotone AXML system. The* semantics *of I, denoted $[I]$, is defined as follows:*

- *either $[I] = J$ for some finite system J such that $I \xrightarrow{*} J$ and the system* terminates *at J;*
- *or $[I] = \cup\{I_i\}$ for some infinite fair rewriting $I \xrightarrow{v_1} I_1 \ldots \rightarrow \ldots \xrightarrow{v_i} I_i \ldots$, i.e each document name in I is assigned the least upper bound of the corresponding documents in the rewriting.*

The semantics of a document $d \in I$, denoted $[d]$, is given by $[I]$. Similarly, for a document d not in I, but which uses only functions in I, the semantics of d can be given by that of the system I augmented with d. The latter will be useful in the sequel to define the semantics of queries, since their results are new documents which are not in I.

As service invocations may bring new data and function nodes, a rewriting may never terminate, as illustrated next.

Example 2.1 *Consider a system having a document d/ a{**f**} with a function **f** that always returns the tree a{**f**}.*
One can show that the only rewriting for this system is:
$d/a\{a\{\boldsymbol{f}\}, \boldsymbol{f}\}$
$d/a\{a\{a\{\boldsymbol{f}\}, \boldsymbol{f}\}, \boldsymbol{f}\}$
$d/a\{a\{a\{a\{\boldsymbol{f}\}, \boldsymbol{f}\}, \boldsymbol{f}\}, \boldsymbol{f}\} \ldots$
*Observe that once some occurrence of **f** has been invoked, it is useless to invoke it again. The infiniteness here is caused by the explicit introduction, at each step, of a new function node. We will see an example where repeated activations of* the same *call brings infinitely new data in the sequel.*

Observe that infinite systems naturally model real world situations, such as subscriptions that keep sending new data to a user. We will see further that, while termination analysis is intricate, it can nevertheless be detected (and even enforced) under certain conditions.

We next show that the semantics is well-defined, i.e. independent of the order of function invocations. As a first step, we prove that any information that can be derived in one particular sequence will also be eventually derived in any other sequence.

Proposition 2.2 *Let I be a system, and suppose $I \stackrel{*}{\rightarrow} J$ and $I \stackrel{*}{\rightarrow} K$. Then, (i) if J terminates at J', then $K \subseteq J'$; and (ii) if $J \stackrel{v_1}{\rightarrow} J_1 \stackrel{v_2}{\rightarrow} J_2 \stackrel{v_3}{\rightarrow} \ldots$, is an infinite fair rewriting, then for some i, $K \subseteq J_i$.*

Proof: (Sketch) The proof works by induction on the number of function invocations in the sequence generating K. It follows from the fact that the rewritings are (1) monotone w.r.t $\subseteq$, and (2) fair. Thus, for every function invocation in K, either all the data it generates already belongs to the "current" system (J, at the induction base), or an analogous invocation will eventually occur. ■

From the previous proposition, it is easy to show that:

Corollary 2.1 *The semantics of monotone AXML systems is well-defined. Namely, if one rewriting terminates, any rewriting terminates at the same finite system. If one rewriting does not terminate, no rewriting terminates and any fair rewriting produces the same infinite system.*

Observe that in a rewriting sequence, each call is activated repeatedly, in a *pull* mode, until it brings no more new data. An alternative view, adopted in [8], would be to consider that services are *continuous*, i.e., that they react to changes of the documents they are defined on, and *push* to their callers new derived results. In such a model, calls need only be activated once, but the subsequent computation may still be infinite. These two push and pull models are essentially equivalent, and our actual implementation can use both [1].

3 Positive Active XML

We will consider next a particular class of monotone systems called *positive systems*, where services are defined by queries whose definitions are known (in contrast to the black-box semantics of services in general monotone systems). We use this additional knowledge to obtain results on termination and finiteness. But first, we need to introduce the query language of positive systems.

3.1 Positive queries

We consider a query language that corresponds to a monotone conjunctive fragment of XQuery [25]. Intuitively, *positive (AXML) queries* are rules of the form $head$:- $body$. The body performs a selection analogous to the *from* and *where* clauses of XQuery, and contains *tree patterns* that we try to match. The head corresponds to the *return* clause, and contains a tree pattern describing the structure of the result. To simplify the presentation, we first consider a rather restricted language, excluding useful features such as regular path expressions, and present our results in this context. Once this is clear, we will consider some extensions, and in particular the use of regular path expressions, in Section 5.

Queries may use four kinds of variables. The first three, *label*, *function*, and *value* variables correspond respectively to the three kinds of nodes in an AXML tree, i.e., nodes marked with labels, function names, atomic values. The last one, *tree* variables, are used to represent subtrees of the document. The core component of queries is *tree patterns*. A *positive AXML tree pattern* is a subtree of an AXML document where some node labels are replaced by label variables, some function names by function variables, and some atomic values (recall that they are assigned to leaves) by value or tree variables.

Definition 3.1 *A* positive query q *is an expression*

$$r :- d_1/p_1, ..., d_n/p_n, e_1, ..., e_m \quad \textit{where}$$

1. *The d_i's are document names, and r, p_i, for $i = 1, \ldots, n$, are positive AXML tree patterns;*
2. *Each variable occurring in r also occurs in some p_i;*
3. *The e_j, $j = 1 \ldots, m$, are inequalities of the form $x \neq y$, where x, y are label, function, or values variables (not tree variable); and no tree variable occurs twice in the body of the rule.*

A simple *query is a query that uses no tree variables.*

Observe that, because of (3), the language prohibits testing for tree (in)equality. This turns out to be essential for guaranteeing monotonicity.

The following is a simple example of a query (with x being a value variable).

```
songs{x} :- directory{cd{title{x},
                         singer{"Carla Bruni"},
                         rating{"***"}}
```

We distinguish between two possible semantics for a query. First, the *snapshot result* is the result of the query when evaluated on a system *in its current state*, without invoking any of the function calls it contains. By contrast, the (full) *result* of a query represents its result when evaluated on the instance corresponding to the semantics of the monotone system, i.e., when all possible calls have been evaluated. These two notions are formalized next.

Snapshot result of queries A variable assignment μ *respects typing* if it assigns labels, function names, atomic values or trees to label, function, atomic or tree variables, respectively. Given a pattern p, $\mu(p)$ denotes the tree obtained from the pattern by substituting each variable by its corresponding value.

Consider a positive query

$$q = r \ :- \ d_1/p_1, ..., d_n/p_n, e_1, ..., e_m$$

Let I be some monotone AXML system containing the documents $d_1, \ldots, d_n$. Then the *snapshot result* of q on I, denoted $q(I)$, is the forest consisting of all documents $\mu(r)$ such that:

- μ is a variable assignment respecting typing and satisfying the inequalities.
- for each d_i/p_i expression in the body, $\mu(p_i) \subseteq I(d_i)$.

(More precisely, the $q(I)$ forest consists of copies of the $\mu(r)$ over disjoint sets of nodes.)

Example 3.1 *As an example, consider the following two documents d and d':*

```
d / r{t{a{1},b{c{2},d{3}}},          d'/ a{1}
      t{a{1},b{c{3},e{3}}},
      t{a{2},b{c{2},f{6}}}}
```

The document d encodes a binary relation. Let x and y be value variables and z a label variable. The following query projects out the labels of b children in tuples having the same a value as one given in d':

```
z :- d'/a{x}, d/r{t{a{x},b{z}}}
```

Its snapshot result is the forest {c,d,e}. On the other hand, consider the query obtained by replacing z by the tree variable Z. Its snapshot result is the forest: {c{2},d{3},c{3},e{3}}.

As shown in the example, tree variables allow to replicate full subtrees of the documents. By contrast, *simple queries* without such variables, can only copy individual nodes. As we shall see in the sequel, this difference is significant. The expressive power of simple queries is more limited, but in return, their analysis is much simpler. For the snapshot semantics, we have:

Proposition 3.1 *(1) The snapshot semantics of positive queries is monotone, i.e., $I \subseteq J$ implies that $q(I) \subseteq q(J)$. (2) It is not monotone anymore if (in)equalities of tree variables are allowed. (3) The snapshot semantics of positive queries can be evaluated in* PTIME. *(4) For the snapshot semantics, containment of positive queries is decidable, i.e., whether for each I, $q(I) \subseteq q'(I)$.*

We omit the proof for space reasons.

Query result The snapshot result of a query takes only into consideration the data currently present in the document, and essentially ignores the intensional data available via service calls. This kind of snapshot result is not what we really expect as an answer to a query. The *result* of a positive query over a monotone system I, that we denote $[q](I)$, considers *all* the data, extensional as well as intensional, and is defined as follows. If I converges to a finite system $[I]$, then $[q](I) = q([I])$. Otherwise, considering any infinite fair rewriting $I = I_1 \ldots I_i \ldots$, we define $[q](I)$ as $\cup q(I_i)$. Because of the fairness and the monotonicity of the rewriting and the query, one can verify (using essentially the same arguments as for Proposition 2.2) that:

Theorem 3.1 *The result of a positive query over a monotone system is well-defined, i.e., it is independent of the rewriting sequence.*

3.2 Positive systems

So far, we viewed services as "black-boxes". In the remaining of the paper, we consider services defined by positive queries whose declarative definition are known. The systems thereby obtained are called *positive systems*. More precisely, a *positive* AXML system (D, F, I) is defined as in Definition 2.3, except that for every function name $\boldsymbol{f} \in F$, $I(\boldsymbol{f})$ is a positive query using document names in $D \cup \{input, context\}$ and function names in F. A positive system where all functions are defined by *simple* queries (namely queries with no tree variables) is called a *simple* positive system.

Positive systems are particular cases of monotone systems. We only have to explain the behavior of positive service calls. So, consider a a positive system I and a function node n labeled $\boldsymbol{f}$ in a document d , where $I(\boldsymbol{f})$ is a positive query.

$$\begin{aligned} r \text{ :- } & (input/q_1,)(context/q_2,) \\ & d_1/p_1, ..., d_n/p_n, \\ & e_1, ..., e_m \end{aligned}$$

When the function is *invoked*, the corresponding query is evaluated (with $input, context$ and $d_1, \ldots, d_n$ instantiated as defined in Subsection 2.2), and its *snapshot result* is appended to the document as siblings of the node n. Because of the monotonicity of the snapshot semantics of positive queries (see Proposition 3.1), it is clear that the system is monotone.

This is illustrated next by two examples. In the first one, the semantics is finite.

Example 3.2 *Consider the simple positive system I containing the documents d_0, d_1, where d_0 encodes a binary relation:*

$$\begin{aligned} I(d_0) &= r\{t\{1,2\}, t\{2,3\}, t\{3,4\}\} \\ I(d_1) &= r\{\boldsymbol{g}, \boldsymbol{f}\} \\ I(\boldsymbol{g}) &= t\{x,y\} \text{ :- } d_0/r\{t\{x,y\}\} \\ I(\boldsymbol{f}) &= t\{x,y\} \text{ :- } d_1/r\{t\{x,z\}, t\{z,y\}\} \end{aligned}$$

It is easy to see that any fair rewriting converges to a finite system where d_1 contains the transitive closure of the relation encoded in d_0.

Observe that the previous simple positive system computes the fixpoint of a datalog program (a transitive closure). More generally, any datalog program can be simulated by

a simple positive system. This and other connections between positive AXML and extensions of datalog are considered in [8]. In particular, an extension of the optimization technique for datalog, Query-sub-Query [23], for positive AXML is considered there. We now consider a positive system with infinite semantics.

Example 3.3 *Consider the document d'/* $a\{a\{b\},\boldsymbol{g}\}$ *with the function* $\boldsymbol{g}$ *defined as*

$a\{a\{X\}\}$:- $context/a\{a\{X\}\}$.

X is a tree variable, hence the system is not simple. One can show that the only rewriting for this system is:

$d/a\{a\{b\},a\{a\{b\}\},\boldsymbol{g}\}$
$d/a\{a\{b\},a\{a\{b\}\},a\{a\{a\{b\}\}\},\boldsymbol{g}\}$
$d/a\{a\{b\},a\{a\{b\}\},a\{a\{a\{b\}\}\},a\{a\{a\{a\{b\}\}\}\},\boldsymbol{g}\}$
...

Here, the same function call returns more and more data. One can verify that the limit is the infinite document where the root a has a single function node child labeled $\boldsymbol{g}$*, and infinitely many distinct subtrees $a^i\{b\}$ for $i \geq 1$.*

The service in this second, non terminating, example is defined by a query with a tree variable. However, non termination may even occur for simple positive systems, i.e. systems where the service queries contain no tree variable; see for instance Example 2.1, whose single service can be defined by the simple query $a\{\boldsymbol{f}\}$:-. We will see below that termination analysis is intricate in general. To understand why, let us first consider the expressive power of AXML.

Expressiveness and completeness AXML computations are *generic* in the sense that they do not interpret atomic values, labels and function names. AXML systems are not complete in the Turing sense; indeed they are not even complete in the relational sense because they are monotone. On the other hand, they can encode very complex computations using recursion and calls to simulate object creation in the style of IQL [7]. One can show:

Theorem 3.2 *Given a monotone, r.e., boolean function f over AXML trees with finite alphabet, there exists a positive AXML system with a service whose result's semantics is the same as that of f.*

Proof: (Sketch) We first show that, given a Turing machine, one can build a positive AXML system that, for each "line" tree (e.g., $\#\{a_1\{a_2\{...a_n\{\#\}\}\}\}$) that encodes an input tree, simulates the run of the Turing machine on this encoding, with the machine tape also encoded by two line trees (for the right and left hand-side of the head). Then, instead of generating a single line encoding of the input tree, we build a system that generates the line encodings of infinitely many trees equivalent to sub-trees of the input, then simulate the Turing machine run on each, unioning the results. Because of monotonicity, this yields the same result as for the single full encoding. ■

Note that genericity is not an issue here because of the finiteness of the alphabets. It is open whether AXML systems capture the generic, monotone and recursively enumerable functions.

As a consequence of the above proof, since AXML can simulate Turing machines, it follows that:

Corollary 3.1 *One cannot decide if a positive system terminates.*

We next consider two particular kinds of systems. For the first one, namely acyclic systems, termination is always guaranteed. For the second, namely simple positive systems, non-termination will be manageable. Both turn out to be quite useful in practice.

Acyclic systems We next define the auxiliary notion of dependency graph and the concept of acyclic systems.

Definition 3.2 *Consider the dependency graph whose nodes are document and function names of the system, and whose edges are defined as follows. Let d be a document name and $\boldsymbol{f},\boldsymbol{g}$ be function names,*

- *there is an edge $(d,\boldsymbol{f})$ if $\boldsymbol{f}$ occurs in $I(d)$.*
- *there is an edge $(\boldsymbol{f},d)$ (resp. $(\boldsymbol{f},\boldsymbol{g})$) if d (resp. $\boldsymbol{g}$) occurs in $I(\boldsymbol{f})$.*

A system is acyclic *if its dependency graph is acyclic.*

It is easy to see that acyclic systems always terminate. In such systems, functions and documents can be totally ordered (based on the dependency graph), starting from the documents and functions that depends on no other document or function. Then one can start evaluating the functions based on this order. Interestingly, each function node has to be invoked only once, since reevaluating it would not produce more data.

Simple positive systems Consider the system in Example 2.1, with its single service defined by a simple query as above. While its semantics is infinite, the resulting tree is *regular*. A *regular* tree t is a (possibly infinite) tree where the number of distinct subtrees of t is finite (up to isomorphism). Such trees can be represented by finite graphs [14]. We will say that a system is *regular* if all its trees are regular. We show next that if a positive system is simple, its (possibly infinite) semantics is regular. By contrast, non simple systems may generate non regular trees. An example is the infinite tree of Example 3.3.

Lemma 3.1 *For every simple positive system I, $[I]$ is regular; and a finite graph representation of $[I]$ can be computed in* EXPTIME.

Proof: (Sketch) The crux is the observation that (1) for all nodes v in $[I]$, each child subtree either belongs to the original system I or is a (rewriting of) the instantiated head of some service query, and (2) identical instantiations, even when located in different places of a document, have equivalent rewritings, providing that their root is not labeled by a function name. (For functions, a somewhat more delicate analysis is required.) Since in simple positive systems rules contain only label/data/function variables, the number of possible instantiations is at most exponential in the size of I, and so is the number of distinct subtrees. The finite graph representation of the result is obtained by recording the instantiations that have already been returned by previous calls, and pointing to their root when the same answer is returned again, rather than constructing a new subtree. ■

The graph representation of the (possibly infinite) semantics of simple positive systems eases their analysis. For instance, one can decide termination for such systems:

Theorem 3.3 *For simple positive systems, termination is decidable in* EXPTIME*, and the problem is* CO-NP *hard.*

Proof: (Sketch) The CO-NP hardness is proved by reduction from the non-satisfiability problem of 3NF formulas. An exponential algorithm to decide termination can be derived from an analysis of the graph representation of Lemma 3.1. ■

3.3 Querying positive systems

We considered above termination for AXML systems. To conclude this section let us consider the (full) result of queries over such systems. We will say that a system I is q-finite if $[q](I)$ is finite. Note that $[I]$ may be infinite and the system still be q-finite. Since the query is defined on the system semantics, detecting query finiteness may be as difficult as detecting system termination.

Proposition 3.2 *For a non-simple query q, (1) one cannot decide if a positive system I is q-finite, (2) acyclic systems are q-finite, (3) for a simple positive system I, deciding q-finiteness is* CO-NP *hard (in the size of I), and can be done in* EXPTIME.

When a query q is *simple*, its result is always finite. This is because the number of possible assignments of each variable in the body of the query is bounded by the size of the (original) system. Nevertheless, when the system itself is not simple, an effective construction of this finite result may be impossible, as it requires knowing the systems semantics. This is illustrated by the following Proposition.

Proposition 3.3 *The problem of testing for a (non simple) positive system I and a simple query q the emptiness of $[q](I)$ is undecidable.*

4 Lazy query evaluation

To answer a query, a naive algorithm may attempt to first fully expand the documents in I, i.e., compute $[I]$, then evaluate q over the resulting documents. As we will see, this simplistic approach suffers from serious drawbacks which may be overcomed using lazy query evaluation.

We first briefly argue that it makes sense to use intensional documents to answer queries. Suppose someone wants to know the rating of Billy Holiday's "*Body and Soul*" song. A possible answer is "****" (assuming that this is what the ***GetRating*** service returns). But it may be preferable to simply answer ***GetRating***{*"Body and Soul"*}. Both answers entail the same information, but the latter delegates the task of invoking the service to the receiver of the result. Formally, we will say that an AXML document α is a *possible answer* to q if it has the same semantics as q's result, or more precisely, $[\alpha] = [[q](I)]$. In that sense, both "****" and ***GetRating***{*"Body and Soul"*} are possible answers.

The choice of which data to materialize and which to leave intensional may be influenced by various parameters, such as performance, communication cost, or security considerations. In [20], typing is used to decide whether particular data should be materialized or not. This issue will not be considered here. The focus here is on query evaluation. We will see that a lazy expansion presents many advantages for query evaluation.

The naive algorithm that first fully expands all the documents would materialize lots of unneeded information. First, it may materialize information in parts of the documents that are irrelevant to the query. Second, it may invoke a call when it suffices to just keep it in the answer (e.g. evaluate ***GetRating***{*"Body and Soul"*} although it is not essential to). In both cases, this may result in attempting to materialize an infinite quantity of information, and consequently lead to a non-terminating computation, although there may exists a finite computation of a possible answer.

This said, the problem becomes one of *lazy evaluation*, i.e., how to expand the document just enough to obtain a possible answer. We will say that a set of function nodes N is *q-unneeded* if the query may be answered even if calls to functions in N are blocked (see formal definition further). Note that if we have detected that the set of *all function nodes* in the system are q-unneeded, then we have identified a point where enough data has been gathered, and no more calls need to be invoked to answer the query. We then say that the system is *q-stable*.

We will see next that these properties are undecidable in the general case, and very expensive to check for simple systems. A practical implementation would thus need to rely on heuristics.

But first, we define formally these notions. To do so, we use the following notation. For a system I and a set of function nodes N in I, let $[I\downarrow_N]$ denote the limit of a (possibly

infinite) rewriting that never invokes calls in N, and is fair for all other function occurrences. As before, let $[q](I{\downarrow_N})$ be the union of $q(I_i)$, for all I_i in the rewriting sequence. Using the same proof as for Proposition 2.2, one can show that $[I{\downarrow_N}]$ and $[q](I{\downarrow_N})$ do not depend on the particular sequence that was chosen (assuming fairness for all nodes but those in N).

Definition 4.1 *Given a system I and a query q over I, a set of function nodes N occurring in I is* q-unneeded *if $[q](I{\downarrow_N})$ is a possible answer to q.*

I is q-stable *if the set of all its function nodes is* q-unneeded.

Observe that the notion of being *unneeded* is subtle. It may be the case that some unneeded call v does produce useful information, but is not needed because some other calls provide this same information. Note that, in particular, being unneeded is not closed under union, i.e., it is possible that N and N' are q-unneeded but $N \cup N'$ is not.

We can show the following.

Theorem 4.1 *(1) One cannot decide, given a positive AXML system I, a query q, a set N of function nodes in I, and a document d whether (i) d is a possible answer to q, (ii) N is q-unneeded, (iii) I is q-stable.*
(2) For simple systems, the three problems are decidable even if q is a non simple query. They are CO-NP *hard w.r.t I, even for simple queries, and can be solved in* NEXPTIME.

Proof: (Sketch) The undecidability and CO-NP hardness proofs use constructions similar to the ones of Corollary 3.1 and Theorem 3.3. The NEXPTIME algorithms are based on building finite graph representations of $[d]$, $[[q](I)]$ and $[[q](I{\downarrow_N})]$, and comparing them. ■

The undecidability should not come as a surprise, since positive AXML systems can simulate Turing machines. The decidability is more interesting. Indeed, decidability can be shown for other properties as well. For instance, we may consider properties linked to the order of call invocations. Clearly, the order is essential if we want a minimal rewriting. In particular, one would like to compute a rewriting of minimal length. One can show that this problem is also decidable (but very expensive) for simple systems. This is omitted here for space reasons.

Weaker properties Recall that our original motivation for studying these properties was to improve execution time, by calling as few services as possible. The previous results show that an exact analysis may be as expensive as the full query computation. Instead, one can use some sufficient "weak" corresponding properties. Intuitively, these weak properties ignore the semantics of functions and just view them as monotone independent black-boxes. One can show that the weak notions, e.g. "weak stability", are sufficient to guarantee their corresponding properties (e.g., weak stability implies stability) and that they are decidable in PTIME. We omit this here for space constraints. In a Web context, these weak notions are very relevant since we often have no way of knowing the code of many Web services that we use, i.e., they indeed behave as black-boxes.

Fire-once semantics To conclude this section, let us highlight connections between the notion of query stability and an alternative semantics for positive AXML systems, where each service call is invoked just once, returning a single answer. In this case, one would like, before answering a call to a service defined by a query q, to decide whether the system is q-stable (or weakly stable in a Web context where some services are black boxes). One can define formally a *fire-once* semantics which considers rewritings where only services for which the system is stable can be invoked. (This may correspond in practice to answering a service only when one knows that the entire answer has been obtained.) One can show that this semantics is well-defined, but may not allow to derive as much data as with the positive semantics. For instance, in Example 3.2, the fire-once semantics would not compute the transitive closure. (The recursive rule will not be evaluated.) In restricted cases, e.g., acyclic systems, the fire-once and the positive semantics coincide.

5 Extending the query language

We considered so far a very restricted query language. This was primarily to simplify the presentation. In the general case, positive AXML systems are difficult to analyze. So, of particular importance, are extensions of simple systems that leave us in subclasses with desirable properties.

To extend the query language, one could consider introducing any monotone query language feature. In particular, one could consider introducing features from popular query languages for XML, assuming they are monotone.

To see an obvious candidate, consider regular path expressions. In our query language, the path expressions in the tree patterns are very restricted, only constant labels are used. Now consider the extension of the query language obtained by allowing instead of labels in tree patterns, to use regular expressions. The interpretation is that there must be a path going down the document tree such that the sequence of labels forms a word in the regular language of the path expression.

We call a positive query that uses regular path expression a *positive+reg query*. Systems whose services are defined by positive+reg queries are called *positive+reg* systems. Since regular expressions can be simulated by deductive programs (which, as mentioned in Section 3, can be expressed by a positive system), any positive+reg system can be simulated by a positive one. However, an issue is that such a simula-

tion requires introducing non-simple features, i.e., using tree variables to copy subtrees. It is possible however to find a more complex simulation that does not require it:

Proposition 5.1 *There is a* PTIME *translation ψ such that, for each positive+reg system I and each positive+reg query q,*

1. *$\psi(I, q) = (I', q')$ where I', q' are positive system and query.*
2. *I', q' are simple if I, q are.*
3. *$[q](I) = [q'](I')$; and in particular, I is q-finite iff I' is q'-finite.*
4. *I is q-stable iff I' is q'-stable.*

Furthermore, ψ also provides a mapping over function nodes such that for each set N of function nodes in I, N is q-unneeded (in I) iff $\psi(N)$ is q'-unneeded (in I').

Proof: (Sketch) For each regular path expression we consider the automaton of the corresponding regular language. The idea is to add to the documents (1) nodes that represent the states of the automaton potentially relevant for each node in the document, and (2) calls to services that, based on the former, compute (backwards) the transitions of the automaton, adding new state information.

For each automata move $\delta(q, a) = p$, the corresponding service is defined by a query that tests if the given (context) node has a child of state p and whose label is a, and if so returns the state q. When the function is invoked this state is stored in the tree. So the states propagate upward in the tree. (To start the computation the final state is stored in all nodes of the tree.) Along with the propagation of states, the services also propagate up the label of the node at the end of the path (for simple I, q) or the node's subtree (for non simple services/queries). ■

To conclude this section, we briefly consider another important feature of query languages for trees, namely nesting. Consider for instance a binary relation:

$d/r\{t\{a\{1\}, b\{2\}\}, t\{a\{1\}, b\{3\}\}, t\{a\{2\}, b\{2\}\}\}$.

Suppose we want to nest it on the a-column to obtain:

$d/r\{t\{a\{1\}, b\{2\}, b\{3\}\}, t\{a\{2\}, b\{2\}\}\}$

This can be achieved with a simple system containing the document d above plus a document d' and functions $\boldsymbol{f}$,$\boldsymbol{g}$:

$d'/r\{\boldsymbol{f}\}$

$\boldsymbol{f}$: $t\{a\{x\}, \boldsymbol{g}\}$:- $d/r\{t\{a\{x\}\}\}$

$\boldsymbol{g}$: $b\{y\}$:- $context/t\{a\{x\}\}, d/r\{t\{a\{x\}, b\{y\}\}\}$

In the example, we compute nesting using a simple system. In general, nesting seems non-simple. For instance, suppose that the b components are complex. Then the $\boldsymbol{g}$ rule has to be turned into a non-simple rule by replacing y by a tree variable Y. This is a case where a non-simple rule does not bring us out of the realm of regular languages. One can find in [7] a powerful language over trees where the representation remains regular; the crux there is the use of strict typing. More work is clearly needed along this line for obtaining systems with more powerful query languages but where important issues such as stability remain decidable.

6 Conclusion

We have presented formal foundations for positive AXML systems. We also showed that the systems obtained by disallowing the copying of trees, namely the simple systems, present nice properties.

The positive AXML model is closely related to models of complex objects and object databases; a survey of the topic may be found in [6]. The AXML model is also in the spirit of previous functional approaches to databases, e.g., [13] and, to a lesser extent, also of object databases [10]. We mentioned also a connection with deductive databases. A main difference with languages that mix complex objects and deduction, e.g., [5], is the semistructured data model (namely, XML) with much less emphasis on typing. There exists an abundant literature on query languages for XML and trees that lead to XQuery; some references may be found in [4].

Many references on the completeness of query languages may be found in [6]. We mentioned the issue of the expressive power of positive AXML that remains open. Recent works on querying the Web combined standard logical querying with Web style navigation [9, 22]. Their computation models are somewhat closer to automata. It would be interesting to investigate whether AXML systems can be simulated by such models and compare the expressive powers.

AXML systems are primarily meant to capture some style of data management on the Web and a P2P architecture. In AXML systems, functions calls are invoked repeatedly. This is meant to capture both a pull mode where the client keeps asking for data and a push mode where the server keeps sending data to the client (pub/sub style). This streams of data are in sharp contrast to a more traditional mode where a query is asked and answered once. In that sense, the work presented here presents connections with continuous and stream queries [12].

AXML systems are typically distributed over several peers. Each peer contains some AXML documents and offers AXML services that are used by other peers. A peer often ignores the semantics of services offered other peers (the queries implementing them). This is the typical situation, for instance, for standard Web services. Observe that this affects dramatically the systems analysis. Consider the detection of termination. Each peer may know that it has reached a fixpoint but a distributed termination detection mechanism is needed to detect termination for the global system. Termination detection in a distributed setting as well as an optimiza-

tion technique for lazy query evaluation (along the lines of QuerySubQuery) in that setting are proposed in a companion paper [8]. This also suggests revisiting notions such as stability, adding to them a dimension of distributed knowledge, e.g., [15].

To conclude, we should observe that the positive AXML systems considered here are very restricted compared to (arbitrary) AXML systems [1]. First, AXML systems allow nonmononotone queries as well as updates. So, confluence is not guaranteed anymore. Also, the model is based on ordered trees (vs. unordered here). Finally, AXML systems offer more control of the invocation of calls that bring them closer in spirit to active databases. For instance, one can specify that a call should be activated periodically (daily) or when a certain event occurs. Aspects related to distribution and to non-monotonicity are important in practice and deserved to be formally studied as well.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and web services integration (demo). In *Proc. of VLDB*, 2002.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. Towards a flexible model for data and web services integration. proc. Internat. Workshop on Foundations of Models and Languages for Data and Objects, Italy, 2001.

[3] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of ACM SIGMOD*, 2003.

[4] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 1999.

[5] S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM TODS*, 16(1), 1991.

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

[7] S. Abiteboul and P . C. Kanellakis. Object identity as a query language primitive. In *Proc. of ACM SIGMOD*, 1989.

[8] S. Abiteboul and T. Milo. Web Services meet Datalog. Submitted to PODS, 2004.

[9] S. Abiteboul and V. Vianu. Queries and computation on the web. *Theoretical Computer Science*, 239(2), 2000.

[10] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of DOOD*, 1989.

[11] The Active XML Website. *http://www-rocq.inria.fr/verso/Gemo/Projects/axml/*.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of ACM PODS*, 2002.

[13] P. Buneman, R. E. Frankel, and R. S. Nikhil. An implementation technique for database query languages. *ACM TODS*, 7(2), 1982.

[14] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, volume 16, pages 231–251. Academic Press, London, 1982.

[15] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[16] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of FOCS*, 1995.

[17] Jelly: Executable XML. *http://jakarta.apache.org/commons/sandbox/jelly*.

[18] The Kazaa file-sharing system. *http://www.kazaa.com*.

[19] Macromedia Coldfusion MX. *http://www.macromedia.com/software/coldfusion/*.

[20] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Dang Ngoc. Exchanging intensional XML data. In *Proc. of ACM SIGMOD*, 2003.

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

[22] M. Spielmann, J. Tyszkiewicz, and J. Van den Bussche. Distributed computation of web queries using automata. In *Proc. of ACM PODS*, 2002.

[23] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. 1st Int. Conf. on Expert Database Systems*, 1986.

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

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