Designing, Specifying and Querying Metadata for Virtual Data Integration Systems

Leopoldo Bertossi[1] and Gayathri Jayaraman[2]

[1] School of Computer Science,
Carleton University, Ottawa, Canada
and
University of Concepcion, Chile
bertossi@scs.carleton.ca
[2] Dept. Systems and Computer Engineering
Carleton University, Ottawa, Canada
gayatri_jraman@yahoo.com

Abstract. We show how to specify and use the metadata for a virtual and relational data integration system under the local-as-view (LAV) approach. We use XML and RuleML for representing metadata, like the global and local schemas, the mappings between the former and the latter, and global integrity constraints. XQuery is used to retrieve relevant information for query planning. The system uses an extended inverse rules algorithm for computing certain answers that is provably correct for monotone relational global queries. For query answering, evaluation engines for answer set programs on relational databases are used. The programs declaratively specify the legal instances of the integration system.

1 Introduction

Current day computer applications have a need to access, process, and report, and specially integrate data from various and disparate sources. Those data integration systems aim to provide a single unified interface for combining data in various formats from those multiple sources [5]. Virtually integrating heterogeneous data for query answering is still an ongoing research challenge.

A common approach uses a mediation system [18] that offers a query interface over a *single global schema*. This global schema consists of relational predicates, in terms of which the user can pose queries. However, there is no actual data contained in them. When the mediator receives a query, it produces a *query plan* that identifies the relevant data sources and the relevant data in them, and specifies how the data obtained from them has to be combined to build the final answer. To produce such a plan, the mediator stores and processes certain *mappings* that associate the predicates in the global schema with those in the local sources.

Some of the challenges in designing a mediator system are: (a) Using standard and expressive enough languages, and formats, for representing all the metadata contained in the mediator; (b) Using a standard way to query the metadata to extract the relevant information; (c) Developing a general query planning mechanism that uses the relevant metadata; and (d) Developing a methodology for executing those plans.

A. Hameurlain and A M. Tjoa (Eds.): Globe 2009, LNCS 5697, pp. 72–84, 2009.

There are different approaches to virtual data integration, depending on how the mappings are represented. The *local-as-view* (LAV) approach, which we follow in this paper, consists in defining the local relations as views over the global schema [12, 4]. In this way, each relevant source relation can be defined independently from other source relations. By doing so, it is easier for any source to join or leave the system, without affecting other source definitions.

Example 1. Consider a relational data source, ***animalkingdom***, containing data about animals. It contains the relation **V1** with attributes *Name, Class, Food*. Another data source, ***animalhabitat***, contains data about animals and their natural habitat. This data is available in the relation **V2** with attributes *Name, Habitat*.

V1	*Name*	*Class*	*Food*
	dolphin	mammal	fish
	camel	mammal	plant
	shark	fish	fish
	frog	amphibian	insect
	nightingale	bird	insect

V2	*Name*	*Habitat*
	dolphin	ocean
	camel	desert
	frog	wetlands

An information system designer interested in providing information on animals defines the following global schema $\mathcal{G}$: *Animal(Name, Class, Food)*, *Vertebrate(Name)*, *Habitat(Name, Habitat)*. This can be done even before the data sources **animalkingdom** and **animalhabitat** (and possibly others) are available to the system. The global relations *Animal* and *Vertebrate* are associated with the local relation **V1** via a Datalog query which defines **V1** as a view over $\mathcal{G}$, as containing animals that are vertebrates:

$$\mathbf{V1}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name). \quad (1)$$

Another mapping describes **V2** as containing animals and their habitat:

$$\mathbf{V2}(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (2)$$

Now consider a Datalog query, $\Pi(\mathcal{Q})$, posed to the mediator, to get all animals with their names and habitat:

$$Ans(Name, Habitat) \leftarrow Animal(Name, Class, Food), Habitat(Name, Habitat). \quad (3)$$

This is a conjunctive Datalog query whose answers cannot be computed by a simple direct computation of the rule body, because the data is not stored as material relations over the global schema. Instead, the mappings that describe the source relations have to be used; to produce a query plan that eventually queries the local sources, where the data is stored. □

In this paper we describe the methodology that is followed in the design of a general system, the *Virtual Integration Support System* (*VISS*), that can be used to specify and use specific mediator-based data integration systems under LAV. The system can be used to integrate multiple relational data sources (or sources wrapped as relational). It uses a standard format, based on XML and RuleML, for representing metadata. More specifically, data about the schemas is represented in and stored as native

XML: (a) The access parameters for the data sources (userid, password, etc.); (b) The structure of the relations at the sources; and (c) The structure of relations in the global schema. The mappings between the global schema and the local schemas are represented in *VISS* using RuleML [7], which is an XML-based markup language for the representation and storage of rules that are expressed as formulas of predicate logic. In order to gather the information needed to compute a query plan, the system uses XQuery, to query the metadata. *VISS* also supports the creation of query plans and their evaluation.

For query planning, *VISS* uses the EIRA algorithm, which is an extended version, introduced in [3, 4], of the *inverse-rules algorithm* (IRA) [9]. EIRA inherits the advantages of the IRA algorithm, but it can handle all the monotone queries, including those with built-ins. We assume that view definitions, i.e. of the source relations, are given by conjunctive queries. A resulting query plan is expressed as an extended Datalog program with stable model semantics [10]. The information gathered from the metadata is used to build this program. In the rest of this paper we illustrate the complete process using our previous and running example.

This paper is structured as follows. Section 2 introduces basic definitions related to virtual data integration, and the Extended Inverse Rules Algorithm. Section3 describes the architecture of *VISS*. Section 4 shows how to use XML and RuleML to specify schemas and mappings. Section 5 describes how XQuery is used to extract the relevant information from the metadata. Section 6 provides a detailed explanation of the query answering mechanism in *VISS*. Section 7 presents some conclusions and ongoing work.

2 Preliminaries

In general terms, a virtual data integration system has three main components: (a) A collection of local data sources with a (union) schema $\mathcal{S}$; (b) A global schema $\mathcal{G}$; and (c) A set of mappings $\mathcal{M}$ between the global and source schemas. A data source is an autonomous database that adheres to its own set of integrity constraints (ICs). In Example 1, **V1**, **V2** are predicates in the source schema $\mathcal{S}$. Those predicates offered by the global schema $\mathcal{G}$ do not have corresponding material instances. In Example 1, *Animal*, *Vertebrate* and *Habitat* are elements of the global schema. In the following we will also denote the integration system with $\mathcal{G}$.

Example 1 shows that it is possible to define other sources contributing with information about animals, such as invertebrates. In this sense, the information in the sources **animalkingdom** and **animalhabitat** can be considered as incomplete with respect to what $\mathcal{G}$ might potentially contain. More precisely, consider that we have a definition $V(\bar{x}) \leftarrow \varphi_i^{\mathcal{G}}(\bar{x}')$, with $\bar{x} \subseteq \bar{x}'$, of a source relation V as a view of $\mathcal{G}$, and a material extension, say v for V. If we have an instance D for $\mathcal{G}$, the view V gets an extension $V[D]$ over D. This instance D is considered to be a *legal instance* of the integration system if $V[D] \supseteq v$, for each view V. This reflects the openness assumption about the sources. $Legal(\mathcal{G})$ denotes the class of legal instances.[1]

[1] In order to simplify the presentation, we will assume that sources are all open. However, we could easily deal with closed and exact sources [4].

Example 2. Consider the extensions for the source predicates: $v_1 = \{(dolphin, mammal, fish), (camel, mammal, plant), (shark, fish, fish), (frog, amphibian, insect), (nightingale, bird, insect)\}$; and $v_2 = \{(dolphin, ocean), (camel, desert), (frog, wetlands)\}$. And the global instance D_0: $Animal = \{(dolphin, mammal, fish), (camel, mammal, plant), (shark, fish, fish), (frog, amphibian, insect), (nightingale, bird, insect), (snake, reptile, frog)\}$; $Vertebrate = \{dolphin, camel, shark, frog, nightingale, snake\}$; $Habitat$=$\{(dolphin, ocean), (camel, desert), (frog, wetlands)\}$.

The evaluation of the views on D_0 gives:

$\mathbf{V1}[D_0]$=$\{(dolphin, mammal, fish), (camel, mammal, plant), (shark, fish, fish), (frog, amphibian, insect), (nightingale, bird, insect), (snake, reptile, frog)\}$;
$\mathbf{V2}[D_0] = \{(dolphin, ocean), (camel, desert), (frog, wetlands)\}$.

In this case, $v_1 \subseteq \mathbf{V1}[D_0]$ and $v_2 = \mathbf{V2}[D_0]$. Hence, D_0 is a legal global instance; and all its supersets are also legal instances. □

Now, given a global query $Q(\bar{x})$, i.e. expressed in terms of the global predicates, a tuple $\bar{t}$ is a *certain answer* to Q if for every $D \in Legal(\mathcal{G})$, it holds $D \models Q[\bar{t}]$, i.e. the query becomes true in D with the tuple $\bar{t}$. $Certain_{\mathcal{G}}(Q)$ denotes the set of certain answers [1] to Q.

The Extended Inverse Rules Algorithm for obtaining certain answers from the integration system is based on a specification as a logic program $\Pi(\mathcal{G})$ with stable model semantics of the legal instances of the system: The stable models of the program $\Pi(\mathcal{G})$ are (in correspondence with) the legal instances of $\mathcal{G}$. The specification is inspired by the IRA algorithm [9], which introduces Skolem functions to invert the view definitions. In our case, instead of functions, we use auxiliary predicates whose functionality is enforced in the specification by means of the *choice operator* [11]. Actually, what the program specifies is the collection of *minimal legal instances*, those that do not contain a proper legal instance. This is because these instances are used to restore consistency of the system for doing consistent query answering (CQA) [3, 4]. (Cf. [2] for a survey of CQA). For monotone queries, using all the legal instances or only the minimal ones does not make a difference. The program $\Pi(\mathcal{G})$ contains the following rules:

1. The facts: $dom(a)$, for every constant $a \in U$; and $V(\bar{a})$ whenever $V(\bar{a}) \in v$, for some source extension $v \in \mathcal{G}$.
2. For every view (source) predicate V in the system with definition $V(\bar{X}) \leftarrow P_1(\bar{X}_1), ..., P_n(\bar{X}_n)$, the rules: $P_j(\bar{X}_j) \leftarrow V(\bar{X}), \bigwedge_{X_i \in (\bar{X}_j \setminus \bar{X})} F_i(\bar{X}, X_i), \quad j = 1, ...n.$
3. For every auxiliary predicate $F_i(\bar{X}, X_i)$ introduced in 2., the rule that makes it functional wrt the dependency of the last argument upon the first arguments:

$$F_i(\bar{X}, X_i) \leftarrow V(\bar{X}),\ dom(X_i),\ choice(\bar{X}, X_i).$$

The choice operator picks up only one value for X_i for every combination of values for $\bar{X}$. This operator can be eliminated as such, or equivalently, defined using standard rules. This point and the whole program is illustrated at the light of Example 3.[2]

[2] These specifications programs can be modified in order to capture also closed and exact sources [4].

Example 3. Program $\Pi(\mathcal{G})$ contains the facts: $dom(dolphin)$, $dom(mammal), \ldots$, $\mathbf{V1}(dolphin, mammal, fish), \ldots$, $\mathbf{V2}(dolphin, ocean), \ldots$. And the rules:

$$Animal(Name, Class, Food) \leftarrow \mathbf{V1}(Name, Class, Food).$$
$$Vertebrate(Name) \leftarrow \mathbf{V1}(Name, Class, Food).$$
$$Animal(Name, Class, Food) \leftarrow \mathbf{V2}(Name, Habitat), F_1(Name, Habitat, Class), F_2(Name, Habitat, Food).$$
$$F_1(Name, Habitat, Class) \leftarrow \mathbf{V2}(Name, Habitat), dom(Class), chosen_1(Name, Habitat, Class).$$
$$chosen_1(Name, Habitat, Class) \leftarrow \mathbf{V2}(Name, Habitat), dom(Class), not\ diffchoice_1(Name, Habitat, Class).$$
$$diffchoice_1(Name, Habitat, Class) \leftarrow chosen_1(Name, Habitat, U), dom(Class), U! = Class.$$
$$F_2(Name, Habitat, Food) \leftarrow \mathbf{V2}(Name, Habitat), dom(Food), chosen_2(Name, Habitat, Food).$$
$$chosen_2(Name, Habitat, Food) \leftarrow \mathbf{V2}(Name, Habitat), dom(Food), not\ diffchoice_2(Name, Habitat, Food).$$

$$diffchoice_2(Name, Habitat, Food) \leftarrow chosen_2(Name, Habitat, U), dom(Food), U! = Food.$$
$$Habitat(Name, Habitat) \leftarrow V2(Name, Habitat).$$ □

Specification programs like these can be evaluated with the *DLV* system [14], for example. It computes certain answers wrt the skeptical (or cautious) stable model semantics of disjuntive logic programs with weak negation and program contraints.

3 Overview of *VISS* ' Architecture

The *VISS* system is implemented in C++. It uses Oracle's Berkeley DB XML [16], an open source XML database, for storing all the XML documents related to a global schema. All XML documents, including the RuleML mappings, are stored in a container which can be queried using XQuery. The intermediate XML results can be stored in the same container.

When a Datalog query is posed to the mediator supported by *VISS*, the latter analyzes the query, to determine the source relations required to answer it. After that, using XQuery to query the metadata, the access information for those relations and their corresponding databases is obtained. Next, import commands are produced, to read tuples from the chosen source relations and store them as facts. These facts form the extensional database used by the program obtained with EIRA, which becomes the query plan. The program is run as in *DLV* (or *DLVDB* [13]). DLV allows for the evaluation of disjunctive Datalog^{not} programs [10], and provides an easy interface to external

databases using ODBC drivers. The result of this program evaluation is the set of certain answers to the global query.

We assume in the rest of this paper that all the schemas are relational. However, the system can also be used to integrate XML data through the use of ODBC XML drivers, which represent native XML documents as relational tables. The virtual data integration system described in [15] also uses RuleML. However, it follows the *global-as-view* approach, according to which global relations are defined as views over the union of the local schemas [12].

The rest of this section explains some of the components of *VISS* as shown in Figure 1. Other components are explained in more detail in subsequent sections.

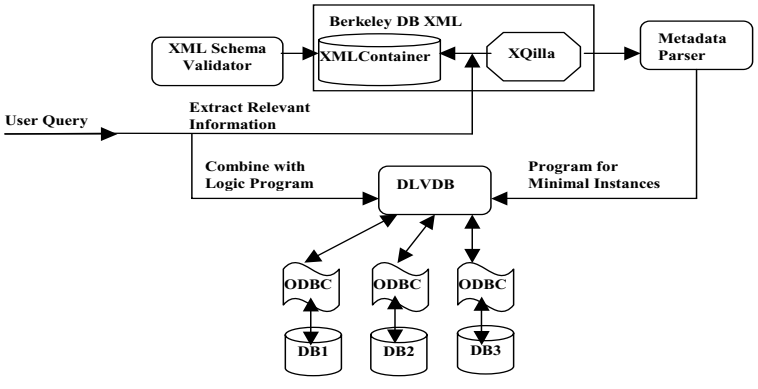

Fig. 1. Architecture of *VISS*

User Queries: *VISS* gives certain answers to monotone queries with built-ins that are expressed in Datalog. Queries in SQL3 format can be translated to Datalog using the SQL3 front end of *DLV*, which is invoked by *VISS*. The query is posed in terms of the predicates in the global schema. The query program is stored in a file named *query,* in the same directory where *VISS* is invoked. (Cf. Section 6 for details.)

XMLSchema Validator: In *VISS* the metadata about the global and source schemas is stored in an XML document; and the mappings are stored in RuleML format. This is described in Section 4. All this metadata is tested using the W3C XML Schema Validator, *XSV*, against the Datalog XML Schema Definition Document and its related submodules (they can be obtained from the RuleML web site). In spite of its name, its has extensions to first-order predicate logic.

Berkeley DB XML: *VISS* uses Oracle Berkeley DB XML 2.4.16 (BDBXML) [16], which is an open source, embeddable XML database/container for storing the XML metadata document. BDBXML includes an XQuery engine. The main BDBXML C++ objects used in *VISS* are *XMLManager*, a high-level class for managing containers; and *XMLContainer*, a file containing XML document. BDBXML uses XQilla 2.0, an open source XQuery, XSLT and XPath implementation that conforms to the XQuery 2.0 W3C

recommendations [16]. *VISS* uses generic XQuery to extract the relevant information from the XML and RuleML metadata documents, to build the logic program specification. The results of the query in XQuery are obtained in the form of an *XMLResults* object.

Metadata Parser and Program Builder: The XML output obtained from the execution of XQuery against the XML metadata is parsed (using the Xerces C++ SAX Parser API calls), and the heads and bodies of the mappings are obtained. It next builds the file containing the program specification for the legal instances.

DLV: The generated logic program, $\Pi(\mathcal{G})$, specifies the minimal legal instances, and is combined with the query program in Datalog, $\Pi(\mathcal{Q})$, and with the *import commands* which load data (the program facts) that are *relevant* to a user query from the source relations. The latter are stored in relational DBMSs, with which *DLV* is able to interact (cf. next item). This combined program is run in *DLV* [13], which is invoked by *VISS*.

Wrappers and Data Sources: *DLV* provides interoperation with relational databases using ODBC connections. The ODBC connection is created as a ODBC Data Source Name (DSN) when a data source is registered with *VISS*. The ODBC connection uses a suitable ODBC driver for the DBMS at a data source. The name and connection parameters for each ODBC connection are also stored in the XML metadata document in *VISS*. Currently *VISS* supports integration of relational data sources.

4 XML and RuleML for the System's Metadata

This section first explains how XML is used in *VISS* for describing the metadata about each of the sources and global schema. For the sources, it stores the type and connection parameters, and the structure of the relations.

All this information is stored using descriptive labels in XML, because it offers the flexibility for defining database schemas that vary from source to source, and also within each of them. All XML documents describing metadata in *VISS* start with a root node **VirInt**. The parameters for connecting to a particular data source are defined by labels such as **Userid**, **Password** and **Databasename**. Relation names in a data source are described using the labels **Rel**, and their attributes are represented using **Var**, as siblings for that **Rel**. The **Source** label is used to list all the data sources participating in the system and their details. The **Global** label is used to list all the details of the global schema, such as its relation names. Listing 1 shows the XML representation of the local and global schema for Example 1.

Example 4. The XML metadata about the data sources **animalkingdom** and **animalhabitat**, and the structure of the local relations **V1** and **V2**, and the global schema consisting of *Animal, Habitat* and *Vertebrate* in *VISS* are shown in Listing 1. □

VISS also contains the mappings between the schemas. They are represented and stored in RuleML format, which is a markup language for representing logical rules. Apart

Listing 1. Sample XML describing the Local and Global Schema in *VISS*

```
<VirInt xmlns:xs="http://www.w3.org/2001/XMLSchema">
<Schema>
    <Local>
        <Source name="animalkingdom">
            <Type>sqlexpress</Type>
            <Hostname>animalkingdom</Hostname>
            <Databasename>animalkingdom</Databasename>
            <Userid>test</Userid>
            <Password>test</Password>
            <Atom>
                <Rel>V1</Rel> <Var>Name</Var> <Var>Class</Var> <Var>Food</Var>
            </Atom>
        </Source>
        <Source name="animalhabitat">
            <Type>mysql</Type>
            <Hostname>animalhabitat</Hostname>
            <Databasename>animalhabitat</Databasename>
            <Userid>test1</Userid>
            <Password>test1</Password>
            <Atom>
                <Rel>V2</Rel> <Var>Name</Var> <Var>Habitat</Var>
            </Atom>
        </Source>
    </Local>
    <Global>
        <Atom>
            <Rel>Animal</Rel> <Var>Name</Var> <Var>Class</Var> <Var>Food</Var>
        </Atom>
        <Atom>
            <Rel>Habitat</Rel> <Var>Name</Var> <Var>Habitat</Var>
        </Atom>
        <Atom>
            <Rel>Vertebrate</Rel> <Var>Name</Var>
        </Atom>
    </Global>
</Schema>
</VirInt>
```

from being similar to and based on XML for representing data, RuleML is also streamlined for storing rules using standard labels, according to the XML Schema Definition for Datalog Rules in the RuleML 0.91 specification [7].

As in the preceding section, the database predicates are represented using **Rel**, and their attributes, using **Var**. Built-ins are represented using **Ind**. **And** indicates that the body of the rule is a conjunction of predicates. **Assert** opens and closes the list of Datalog rules. **Implies** describes the rule is an implication, and hence contains **head** and **body** labels.

Example 5. (example 1 continued) The mappings that are given as view definitions of local relations, **V1** and **V2**, as views of the global schema consisting of $Animal$, $Habitat$ and $Vertebrate$ are represented in RuleML format in Listing 2. □

Currently, the metadata that *VISS* uses, as XML documents, has to be created manually, and made available to *VISS*. However, *POSL* could be used to convert Datalog rules to RuleML [6].

5 Using XQuery to Build the Specification

In order to compute the certain answers to a query, the XML metadata and RuleML mappings have to be queried. After extracting the relevant information, the logic program $\Pi(\mathcal{G})$ that specifies of the legal instances can be generated. It will have the form shown in Example 3. The extraction of the relevant information from the XML and

Listing 2. Sample RuleML describing the mappings in *VISS*

```
<RuleML xmlns:rule="http://www.ruleml.org/0.91/xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ruleml.org/0.91/xsd
file:///C:/thesis/VDI/datalog.xsd">
<Assert>
  <Implies>
    <head>
      <Atom>
        <Rel>V1</Rel> <Var>Name</Var> <Var>Class</Var> <Var>Food</Var>
      </Atom>
    </head>
    <body>
      <And>
      <Atom>
        <Rel>Animal</Rel> <Var>Name</Var> <Var>Class</Var> <Var>Food</Var>
      </Atom>
      <Atom>
        <Rel>Vertebrate</Rel> <Var>Name</Var>
      </Atom>
       </And>
    </body>
  </Implies>
  <Implies>
    <head>
      <Atom>
        <Rel>V2</Rel> <Var>Name</Var> <Var>Habitat</Var>
      </Atom>
    </head>
    <body>
      <And>
      <Atom>
        <Rel>Animal</Rel> <Var>Name</Var> <Var>Class</Var> <Var>Food</Var>
      </Atom>
      <Atom>
        <Rel>Habitat</Rel> <Var>Name</Var> <Var>Habitat</Var>
      </Atom>
       </And>
    </body>
  </Implies>
</Assert>
</RuleML>
```

RuleML metadata is done using XQuery, an SQL-type query language for XML documents. We use a reduced class of queries expressed in XQuery, the *FLWOR* expressions, which are of the form:

```
FOR <var> IN <expr>
LET <var> := <expr>
WHERE <expr>
ORDER BY <expr>
RETURN <expr>
```

FOR and LET clauses select all the tuples and bind the expressions. The WHERE clause specifies the condition for filtering the tuples. The ORDER BY clause gives the sorting order.

The RETURN clause specifies the projected attributes for the result. A generic query in XQuery is presented in Listing 3. It is used to extract, from the Listings 1 and 2, the heads and bodies, including built-ins, that are required to build the relevant view definitions, as Datalog rules.

Example 6. Using the query in XQuery shown in Listing 3, the XML metadata in Listings 1 and 2 can be queried. The rules are obtained in XML format as shown in Figure 2. □

The intermediate information in XML thus obtained in Example 6 is then parsed using a C++ program called *minInstance*. In this way, the program specifying the legal instances shown in Example 3 is constructed. *minInstance* is implemented in C++, to read in an XML file that contains details of each LAV mapping, such as the head and body of each

Listing 3. Querying XML and RuleML metadata with XQuery

```
<!--First navigate to the node using the container in Berkeley DB XML 2.4.16-->
for $n in collection('mappingAlias')/VirInt return
<rules>                                  <!--List all rules-->
    {for $x in $n/RuleML/Assert/Implies
    where distinct-values($x/body/And/Atom/Rel) =
    ("Animal", "Habitat")
    return
    <rule>                               <!--Describe the rule-->
    {for $d in $x return
        <head r1='{$d/head/Atom/Rel}'>   <!--Describe the rule-->
        { for $v in $x/head
        where $v/Atom/Rel=$d/head/Atom/Rel
        return
            concat($d/head/Atom/Rel,'(',
            string-join($v/Atom/Var,','),')')
        }
        </head>
    }
    {for $b in distinct-values($x/body/And/Atom/Rel)
    return
        <body r1='{$b}'>                 <!--Describe the body of the rule-->
        { for $m in $n/Schema/Global/Atom
        where $m/Rel=$b
        return concat($b,'(',
        string-join($m/Rel/following-sibling::Var,','),')')
        }
        </body>
    }
    <body r1=''>                         <!--Describe built-ins in the rule-->
{ for $q in distinct-values($x/body/And/Atom/Ind)
let $l as xs:integer := index-of($x/body/And/Atom/*,$q)-1
let $r := $x/body/And/Atom/Ind/preceding-sibling::Rel/text()
let $s := $n/Schema/Global/Atom/Rel[text()=$r]/../Var[$l]
return
if ($q != '') then
(concat('(',$s,'=',$q,')',','))
else ()
}
    </body>
    </rule>
    }
</rules>
```

```
<rules xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='mappingrule.xsd'>
    <rule>
        <head r1="V1">V1(Name, Class, Food)</head>
        <body r1="Animal">Animal(Name, Class, Food)</body>
        <body r1="Vertebrate">Vertebrate(Name)</body>
    </rule>
    <rule>
        <head r1="V2">V2(Name, Habitat)</head>
        <body r1="Animal">Animal(Name, Class, Food)</body>
        <body r1="Habitat">Habitat(Name, Habitat)</body>
    </rule>
</rules>
```

Fig. 2. Rules obtained with XQuery

rule, as shown in Example 6. It then outputs the specification logic program, but without the ground facts. *minInstance* first parses the XML file using the Xerces C++ SAX Parser API calls, and stores the content of each rule in a simple internal data structure. After reading in all the atoms of a rule (one head and the bodies of the rules defining it), each rule atom is parsed into the predicate and attributes. The strings are then manipulated in C++ using the STL String class functionality, to output the legal instances specification.

6 Query Answering

The current implementation of *VISS* supports the computation of the certain answers to queries. The program $\Pi(\mathcal{G})$, illustrated in Example 3, is combined with a program that generates the facts from the relevant relations using *#import* commands. This combined program is run in *DLV* with the query program $\Pi(\mathcal{Q})$. For monotone queries, the certain answers can be computed in this way [4].

However, for a given query $\Pi(\mathcal{Q})$, only a relevant portion of $\Pi(\mathcal{G})$ is required, the one containing rules that define global predicates that appear in $\Pi(\mathcal{Q})$.[3] Accordingly, first the relevant source relations are identified using a simple C++ module called *relExtract*. It is used to identify the global relations in $\Pi(\mathcal{Q})$. Next, the source relations defined in terms of the latter are derived, using XQuery as shown in Example 7.

Example 7. Consider query (3). Using *relExtract*, we get the predicates *Ans, Animal, Habitat*. XQuery is used against the intermediate XML result obtained in Example 6, to get the relevant source relations:

```
for $a in /rules/rule
where $a/body/@r1 = ("Ans", "Animal", "Habitat")
return data($a/head/@r1)
```

We obtain **V1** and **V2** as the relevant source relations for this query. □

Once the relevant source relations have been detected, the system extracts from the Listing 1 the connection information for those sources, i.e. *hostname*, *databasename*, *userid*. Import commands are generated to extract from the data sources the facts needed by the specification program. They are of the form $\#import(databasename, "username", "password", "query", predname, typeConv)$, where *databasename, username, password* are read from the XML metadata. *query* is an SQL statement that constructs the table that will be imported, *predname* defines the name of the predicate that will be used, and *typeConv* specifies the conversion for mapping DBMS data types to Datalog data types for each column. XQuery is used to simultaneously obtain relevant information and construct the import commands around the former.

Example 8. Running a query in XQuery against the XML metadata in Listing 1, we generate the import commands for the relevant source relations obtained in Example 7:

```
#import(animalkingdom,"test","test","SELECT * FROM V1",
V1, type : Q_CONST, Q_CONST, Q_CONST).
#import(animalhabitat,"test1","test1","SELECT * FROM V2",
V2, type : Q_CONST, Q_CONST).
```

Here, `Q_CONST` is a conversion type specifying that the column is converted to a string with quotes. It can be used in general for all data types. These import commands are combined with the program for the legal instances shown in Example 3. The rules specifying the global relations, plus the import commands, are combined with the query program $\Pi(\mathcal{Q})$ at hand. To obtain the certain answers to query $\mathcal{Q}$, this combined program is run with *DLV*, that provides an interface to the databases at the sources. For the query in (3), we obtain:

[3] We are not considering global ICs. Cf. Section 7.

```
dl.exe -silent -cautious test2.dlv
dolphin, ocean
camel, desert
frog, wetlands
```
□

7 Conclusions

The *VISS* system provides an infrastructure for virtually integrating multiple data sources according to the LAV approach. Concrete integration systems are specified in terms of metadata using a standard XML/RuleML representation. This design allows data sources to be added or removed from the system without affecting other data sources. XQuery is used to extract the relevant information that is needed to create plans for evaluation of global queries. In their turn, these plans are expressed as logic programs with stable model semantics; and they involve a specification of the legal instances of the integration system.

Currently the system does not provide support for global integrity constraints (GICs). Enforcing them on the system is not possible due to the autonomy of the local sources. In consequence, GICs have to be handled as in *consistent query answering* [2]. In our running example, we might have the functional dependency (FD) $Name \rightarrow Class$ satisfied by relations **V1** and an additional **V3**, which is defined by $\mathbf{V3}(Name, Class, Food) \leftarrow Animal(Name, Class, Food), Vertebrate(Name)$. However, there is no guarantee that the same (now global) FD will be satisfied by (the legal instances of) the global relation $Animal(Name, Class, Food)$. In cases like this, it becomes necessary to retrieve those answers that are consistent wrt these GICs, at query time. A characterization of- and mechanisms for this form of consistent query answering in virtual data integration have been introduced in [4].

This addition of support for consistent query answering in *VISS* corresponds to work in progress. Global ICs (and also local if desired) can be easily specified in *VISS* using RuleML 0.91. Admitting global ICs would enhance the functionality of *VISS*, providing support for computation of consistent answers, similar to the approach used in the *Consistency Extractor System* [8], which works on single and possibly inconsistent databases.

References

[1] Abiteboul, A., Duschka, O.: Complexity of Answering Queries Using Materialized Views. In: Proc. ACM Symposium on Principles of Database Systems (PODS 1998), pp. 254–263 (1998)

[2] Bertossi, L.: Consistent Query Answering in Databases. ACM Sigmod Record 35(2), 68–76 (2006)

[3] Bravo, L., Bertossi, L.: Logic Programs for Consistently Querying Data Integration Systems. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 10–15. Morgan Kaufmann, San Francisco (2003)

[4] Bertossi, L., Bravo, L.: Consistent Query Answers in Virtual Data Integration Systems. In: Bertossi, L., Hunter, A., Schaub, T. (eds.) Inconsistency Tolerance. LNCS, vol. 3300, pp. 42–83. Springer, Heidelberg (2005)

[5] Bernstein, P., Haas, L.: Information Integration in the Enterprise. Communications of the ACM 51(9), 72–79 (2008)
[6] Boley, H.: Integrating Positional and Slotted Knowledge on the Semantic Web (2005), http://www.ruleml.org/posl/poslintweb-talk.pdf
[7] Boley, H., Tabet, S., Wagner, G.: Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In: Proc. Semantic Web and Web Services (SWWS 2001), pp. 381–401 (2001)
[8] Caniupan, M., Bertossi, L.: The Consistency Extractor System: Querying Inconsistent Databases using Answer Set Programs. In: Prade, H., Subrahmanian, V.S. (eds.) SUM 2007. LNCS, vol. 4772, pp. 74–88. Springer, Heidelberg (2007)
[9] Duschka, O., Genesereth, M., Levy, A.: Recursive Query Plans for Data Integration. Journal of Logic Programming 43(1), 49–73 (2000)
[10] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems 22(3), 364–418 (1997)
[11] Giannotti, F., Pedreschi, D., Sacca, D., Zaniolo, C.: Non-Determinism in Deductive Databases. In: Delobel, C., Masunaga, Y., Kifer, M. (eds.) DOOD 1991. LNCS, vol. 566, pp. 129–146. Springer, Heidelberg (1991)
[12] Lenzerini, M.: Data Integration: A Theoretical Perspective. In: Proc. ACM Symposium on Principles of Database Systems (PODS 2002), pp. 233–246 (2002)
[13] Leone, N., Lio, V., Terracina, G.: DLV^{DB}: Adding Efficient Data Management Features to ASP. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 341–345. Springer, Heidelberg (2003)
[14] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
[15] Maclachlan, A., Boley, H.: Semantic Web Rules for Business Information. In: Proc. IASTED International Conference on Web Technologies, Applications and Services, pp. 146–153 (2005)
[16] Anatomy of an XML Database: Oracle Berkeley DB XML. An Oracle white paper (2006), http://www.oracle.com/technology/products/berkeley-db/xml/index.html
[17] Ullman, J.: Information Integration Using Logical Views. Theoretical Computer Science 239(2), 189–210 (2000)
[18] Wiederhold, G.: Mediators in the Architecture of Future Information Systems. IEEE Computer 25(3), 38–49 (1992)