Deep Web Queries in a Semantic Web Environment

Thomas Hornung[1] and Wolfgang May[2]

[1] Institut für Informatik, Universität Freiburg
hornungt@informatik.uni-freiburg.de
[2] Institut für Informatik, Universität Göttingen
may@informatik.uni-goettingen.de

Abstract. Access to Deep Web sources is concerned with querying data that is hidden behind Web forms and primarily not accessible by common query languages. Web forms do not contain any type information, and it thus follows that Deep Web sources only work on string data in its rudimentary form. In this paper, we demonstrate how Semantic Web technologies can be used to first lift Deep Web sources to the level of databases with a precise schema and strong typing information and finally to the level of Semantic Web applications. A special focus in this context is on handling measurements, units and dimensions, which is an important issue when data from multiple Deep Web sources is declaratively combined for more involved querying tasks.

1 Introduction

Most of the information that is needed for daily tasks is available on the Web. The main problem is often not the general access to the information, but to access the right information, and to combine it in an appropriate way. Direct query evaluation is not always possible: most of the data is not immediately available for querying, but kept in the *Deep Web* or *Hidden Web*, which consists of dynamically generated result pages of numerous databases, which can only be queried interactively via Web forms. For the *human* user, these Deep Web sources, made visible as HTML pages, have an implicit semantics. For accessing them in an automated environment, this semantics is not available. In contrast to Semantic Web knowledge bases, and even to databases, Deep Web sources have a very primitive data model: their only concept are strings. Even WSDL specifications of (XML-based) Web services provide more information since they have an notion of "answer" and they specify what datatype is returned as response. Current use of Deep Web sources in computerized workflows very explicitly incorporates the background knowledge of a human, e.g., by explicitly programming Web data extraction processes.

For more generic computerized access, Deep Web sources must be *annotated* by metadata. In a first step, this metadata lifts them to the level of databases where the attributes are assigned with datatypes and optionally simple (range)

W. Abramowicz and D. Flejter (Eds.): BIS 2009 Workshop, LNBIP 37, pp. 39–50, 2009.

integrity constraints. On a higher, semantical, level, annotations provide the link to the semantics of an application domain.

Note that one must distinguish between making Deep Web sources machine-*accessible* (which means the tasks of Deep Web navigation to request the hidden data as HTML contents, and to program wrappers to extract data records from these HTML pages) and making them machine-*understandable* which means to lift the extracted data on the level known from databases or even Semantic Web knowledge bases. We build our work on [16] (navigation) and [14] (extraction), that solve the accessibility issue, and we deal with the second issue in this paper.

Structure of the paper. We introduce the MARS framework that provides the environment for informational workflows using Deep Web queries in this paper in Section 2. In Section 3 we discuss annotations. In Section 4, we apply the results to develop an ontology for a comprehensive description of Deep Web sources wrt. the underlying domain ontologies. Section 5 shows how such descriptions are used to embed queries against Deep Web sources in MARS workflows. Section 6 discusses related work, and a conclusion follows in Section 7.

2 MARS: The Framework

The *MARS (Modular Active Rules for the Semantic Web)* Framework [9] provides an open framework for ECA (Event-Condition-Action) rules and for processes. The core of the MARS approach are a model and an architecture for ECA rules that use *heterogeneous* event, query, and action languages. In this paper, we consider one such language, the query language *DWQL (Deep Web Query Language)* that allows to pose queries against Deep Web sources. MARS is an open framework in the sense that arbitrary languages following this metaphor can be embedded; DWQL is such a language.

The MARS data flow through a rule or a process and to/from the processors of the constituents is based on sets of tuples of variable bindings in the style of deductive rules as illustrated in Figure 1. The state of the computation is represented by a set of tuples of variable bindings, i.e., every tuple is of the form $t = \{v_1/x_1, \ldots, v_n/x_n\}$ with $v_1, \ldots, v_n$ variables and $x_1, \ldots, x_n$ elements of the underlying domain (which is in our case the set of strings, numbers, and XML literals). Thus, for given variables $v_1, \ldots, v_n$, such a state can be seen as a relation whose attributes are the names of the variables.

Elements of constituent languages, such as DWQL queries, are represented in the MARS XML markup by elements of the form

```
<dwql:Query xmlns:dwql="http://www.semwebtech.org/languages/2008/dwql#">
  <dwql:view dwql:resource="identifying URI of the DWQL view" />
  <dwql:inputVariable name="x"  ... further annotations ... />
  <dwql:outputVariable name="y"  ... further annotations ... />
  further specification in DWQL markup as element content
</dwql:Query>
```

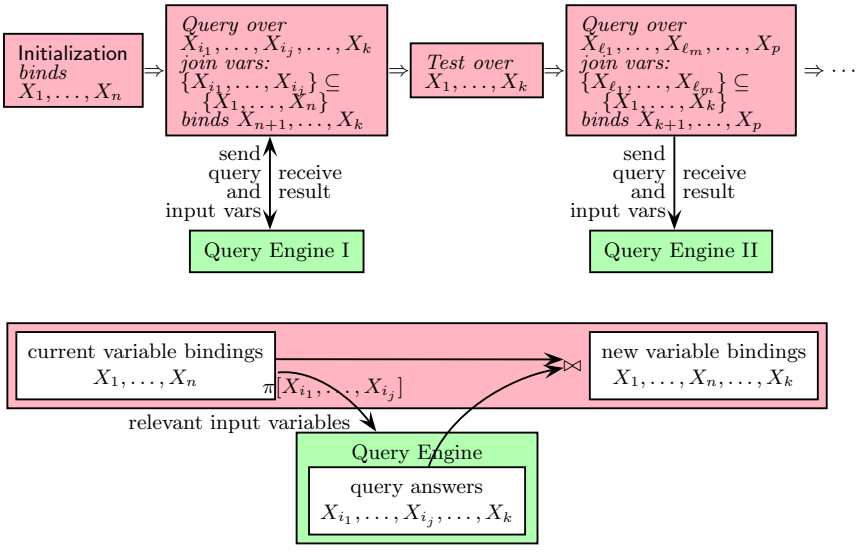

Fig. 1. Use of Variables in MARS

that contain the variable usage characteristics of the constituent. During execution, the selection of the actual services is done by a namespace-based infrastructure [5]. Based on the variable usage specification, only the relevant input variables are submitted together with the language fragment.

3 Literals, Measurements, Dimensions, and Units

Handling and combining queries against autonomous Deep Web sources requires some metadata knowledge about the values to be expected to deal with. Variables can be bound to literal values, and in RDF environments also to URIs (which are represented by strings, but represent the objects).

Schema. For handling values of variables programmatically, knowledge of the *datatypes* is mandatory. The datatypes are the same as in common programming languages and database systems; provided for the RDF world by the XML Schema simple types. In addition to the basic data types like strings and numbers, XML Schema provides `xsd:date`, `xsd:time` and `xsd:dateTime` datatypes similar to SQL. For actual usage, the syntactical representation by some format, e.g., "DD-MM-YYYY" can be specified.

Semantics. On the semantical level, the notion of *dimension* of a property is central: dimensions are e.g., the physical dimensions like *length*, *duration*, *voltage*, but also non-physical dimensions like *distance* (which is physically a length), or *price*. Values of dimensions are actually given by (value-unit)-pairs, like 100 km, or 250 €. Every dimension is associated with a set of units. In contrast to closed

applications, the units may differ between automonous sources (e.g., miles vs. kilometers, or $ vs. € or £); in such cases the conversion factors have to be known. Properties and also variables ranging over the values of a property are annotated with a dimension.

The MARS Ontology for Annotations. In MARS, processes and their constituents can be represented and annotated in RDF [13] and OWL [11]. A fragment of the ontology for dimensions, units, and conversions is shown below in Turtle [17] format. Some conversions are fixed (e.g., miles to kilometers), and some are dynamic, e.g., $ to €; for the latter, google (search e.g. for "100 USD in EUR") is used internally. Currencies are represented by fixed URIs such as <http://www.semwebtech.org/mars/currencies#EUR>.

```
@prefix : <http://www.semwebtech.org/mars#> .
@prefix dim: <http://www.semwebtech.org/mars/dimensions#> .
@prefix unit: <http://www.semwebtech.org/mars/units#> .
@prefix curr: <http://www.semwebtech.org/mars/currencies#> .
dim:Length a :Dimension;
  :hasUnits unit:meter, unit:kilometer, unit:mile, ... .
dim:Price a :Dimension;
  :hasUnits curr:USD, curr:EUR, curr:PLN, ... .
  owl:equivalentClass
  [ a owl:Restriction; owl:onProperty :hasUnits;
    owl:allValuesFrom :Currency] .
[ a :FixedConversion;
  :from unit:kilometer; :to unit:mile;  :factor 1609.3 ] .
[ a :DynamicConversion; :from curr:EUR; :to curr:USD ] .
[ a :DynamicConversion; :from curr:EUR; :to curr:PLN ] .
```

4 Deep Web Source Modeling

Deep Web sources can be considered on two levels: as data sources on the plain *database and (XML) Web level*, and wrt. their domain ontology on the *Semantic Web level*. We associate a precise source capacity description on both levels to each Deep Web source (which has to be done manually for each source).

Conceptually, every Web Data Source can be seen as an n-ary predicate $q(\overline{x}) = q(x_1, \ldots, x_n)$ (its *characteristic predicate*, which contains all input/output mappings). The first modeling step consists of *naming* the variables of this predicate by so-called *tags*. The different interaction patterns with the Web site (e.g. filling out forms, checkboxes, etc.) can be regarded as predefined views over the characteristic predicate. The modeling associates each view with a unique identifying URI (which is not the URL of the corresponding Web form, but "simply" some RDF URI) which is used (e.g. in MARS) for referring to that view. For each view v, its signature is specified in terms of one or more tags declared as input and output arguments. In the remainder of the paper we denote this signature as $\overline{out} \leftarrow v(\overline{in})$, where $\overline{in}$ and $\overline{out}$ are sets of tags. Each input argument corresponds

to an input element in the Web form, and each output argument corresponds to certain data records in the result page (cf. [7]).

Example 1 (Online Railway Schedule). *The online train schedules of railway companies are a typical example for Deep Web sources. Users can enter a start and a destination, a date and a desired departure or arrival time. The answer contains a list of relevant connections, usually together with prices.*

For the German Railway Web portal at `http://www.bahn.de`*, we tag the source with* start, dest, deptTime, arrTime, desiredDeptTime, desiredArrTime, date, duration, price. *The provided views have the signatures*

(deptTime, arrTime, duration, price) ←
germanRailwaysByDept(start, dest, date, desiredDeptTime) and
(deptTime, arrTime, duration, price) ←
germanRailwaysByArr(start, dest, date, desiredArrTime) .

A result of the first view looks as follows:

germanRailwaysByDept(
(start/"Freiburg", dest/"Göttingen", date/"03.02.2009", time/"08:00")) =
{ (deptTime/"08:57", arrTime/"13:07", duration/"4:10", price/"95.00"),
(deptTime/"09:03", arrTime/"14:48", duration/"5:45", price/"85.00"), ...}

Note that there is e.g. no view to retrieve all cities that can be reached from a given starting point within one hour traveling.

In set-oriented approaches like MARS, the input can consist of multiple tuples. For that, the answer tuples are always assumed to also contain the bindings of all input variables. With this, the results can be joined as shown in the lower part of Figure 1.

So far, there are only strings. Annotations are now made on the tag level, since the same annotations hold for each view over the source.

Datatypes and Units. According to Section 3, each tag is associated to some datatype, optionally to a specific syntactical representation, and a unit. For the specification of the format, MARS uses the one from Java's SimpleDateFormat.

Example 2 (Annotations to the Railway Source). *For the German Railways source, the tags are annotated as follows with datatypes, dimensions, syntactical representation (usually called* format*), and units.*

Tag	Datatype	Format	Unit
start, dest	xsd:string	–	–
deptTime, arrTime, desiredDeptTime, desiredArrTime	xsd:time	"HH:mm"	(internal)
duration	xsd:time	"HH:mm"	(internal)
date	xsd:date	"dd.MM.yyyy"	(internal)
price	xsd:decimal		curr:EUR

Source descriptions of DWQL wrappers for other railway portals have a similar signature, except probably the date format, and the currency: the source description of the analogue for Polish railways, `http://www.pkp.pl`*, differs only in the last entry – the unit of their price is Zloty, denoted by the URI* <*http://www.semwebtech.org/mars/currencies#curr:PLN*>*.*

Relationship with the Domain Ontology. The actual values, which are literals, have been described from the programming and data handling point of view above. From the semantical point of view, some of these literals, namely those that are only strings without dimension or datatype (in the above example: start and dest), denote entities of the according application domain.

Example 3 (Deep Web Source Description for German Railways). *In the railway example, the* start *and* end *tags are annotated to represent cities. The complete DWQL Source Description in RDF (N3) format is given in Figure 2. It lists the the tags used by the source, the views provided by the source, and for each view which tags are used in it as input or output. Note that the tag identifiers of the form "_:xxx" act only internally as identifiers. To the outside, only the tag names are known as illustrated by the SPARQL [15] query*

```
select ?U
where { ?S :providesView <bla://dwql-views/travel/germanRailwaysByDept> .
        ?S :hasTag [ :name "price"; :unit ?U ] }
```

that can be used to query the unit of the "price" slot of answers when retrieving connections by departure time. It yields the URI <*http://www.semwebtech.org/mars/currencies#EUR*>*. Such queries are used when the domains/units of variables of a process that contains a DWQL query are derived; as described in the next section.*

5 Embedding Deep Web Queries in MARS Processes

5.1 Annotation of Processes

The dataflow in MARS rules and processes is organized via tuples of variable bindings as depicted in Figure 1. The variables of MARS processes are optionally also annotated with a datatype and a dimension. This can be done automatically by analyzing the process and its variable usage if the subexpressions (here: DWQL queries) are accordingly annotated.

For annotation with units, either every single value can be annotated (which would require to store the unit as an additional column in the underlying database), or the variable is annotated once (usually based on the annotation of the source where the values originate from), and every value is transformed to that unit. For MARS, we chose the latter alternative.

Note that processes over homogeneous sources, e.g., which all use kilometers and Euro, work well even without explicit annotation. The annotation becomes important when the sources use different units.

```
 @prefix dim: <http://www.semwebtech.org/mars/dimensions#> .
 @prefix curr: <http://www.semwebtech.org/mars/currencies#> .
 @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
 @prefix : <http://www.semwebtech.org/languages/2008/dwql#> .
 @prefix travel: <http://www.semwebtech.org/domains/2006/travel#> .

<bla://dwql-views/travel/germanRailways> a :DeepWebSource;
 :baseURL <http://www.bahn.de>;
 :providesView <bla://dwql-views/travel/germanRailwaysByDept>,
               <bla://dwql-views/travel/germanRailwaysByArr>;
 :hasTag _:start, _:dest, _:deptT, _:arrT,
         _:dDepT, _:dArrT, _:dur, _:date, _:price.

_:start a :Tag; :name "start"; :datatype xsd:string; :denotes travel:City.
_:dest  a :Tag; :name "dest"; :datatype xsd:string; :denotes travel:City.
_:deptT a :Tag; :name "deptTime"; :datatype xsd:time; :format "HH:mm".
_:arrT  a :Tag; :name "arrTime"; :datatype xsd:time; :format "HH:mm".
_:dDepT a :Tag; :name "desiredDeptTime"; :datatype xsd:time; :format "HH:mm".
_:dArrT a :Tag; :name "desiredArrTime"; :datatype xsd:time; :format "HH:mm".
_:dur   a :Tag; :name "duration"; :datatype xsd:time; :format "HH:mm".
_:date  a :Tag; :name "date"; :datatype xsd:date; :format "dd.MM.yyyy".
_:price a :Tag; :name "price"; :datatype xsd:decimal;
                               :dimension dim:price; :unit curr:EUR.

<bla://dwql-views/travel/germanRailwaysByDept> a :DeepWebView;
  :hasInputVariable _:start, _:dest, _:dDepT, _:date;
  :hasOutputVariable _:deptT, _:arrT, _:dur, _:price.
<bla://dwql-views/travel/germanRailwaysByArr> a :DeepWebView;
  :hasInputVariable _:start, _:dest, _:dArrT, _:date;
  :hasOutputVariable _:deptT, _:arrT, _:dur, _:price.
```

Fig. 2. DWQL Source Description for German Railways

Communication with Sources. As the annotations of the sources include the units that are required for the input variables, the values sent to the sources are converted accordingly (wrt. units, and also wrt. the syntactic representation, e.g., in case of time and date). As mentioned above, the returned values are also converted if required.

5.2 Embedding Deep Web Queries in MARS

The basic embedding pattern for queries has been shown in Section 2. For DWQL queries, the pattern has to be filled to contain all relevant information for communication with the DWQL service.

Usually, the variable names used in the MARS process do *not* coincide with the tag names of the DWQL views (in the same way as in programming, the variables in a method call do not coincide with the formal parameters of a method definition). As DWQL views are not positional (which wpuld mean that the arguments are ordered), but *slotted*, the pattern has to indicate how the

MARS variables are mapped to the view's variables/tags (and vice versa for the result variables).

The MARS processing model follows the idea of input and output variables for processing components of rules and processes. Thus, DWQL can be embedded in a homogeneous way. The *variable usage characteristics* of language components, i.e., a profile which variables are used, which have to be supplied as input (logic programming: negative use) and which can be bound by the evaluation of the component (logic programming: positive use) is contained in the process specification as illustrated in the example below.

5.3 Use Case: Combination of Queries against Railway Schedules

The use of the schedule of German Railways as a Deep Web source has been introduced in the above example. For international connections, for instance, from Freiburg to Poznan, the prices are not always returned. A suitable strategy is here to look up connections to the stations near to the border against the railway source in the origin country, and from these stations to the destination in the railway source of the destination country (and analogously for connections that run through three or more countries). Note that the necessity for conversion of prices naturally emerges in this situation.

We illustrate the approach using the above-mentioned connection from Freiburg to Poznan, using `http://www.bahn.de` for German Railways and `http://www.pkp.pl` for Polish Railways as Deep Web sources. The wrappers to both have been implemented based on [16,14].

The full query workflow can be specified in MARS/CCS [10,6] as shown in Figure 3 where we abstract from some parts that are not relevant for the Deep Web issues. The workflow is simplified such that it applies only to travels to direct neighbor countries. We also assume a data source that can be queried for the border stations for each pair of neighboring countries.

First, the variables start, startC, dest, destC, date, and time, are bound to their initial values, resulting in the single tuple

(start/"Freiburg", startC/"D", dest/"Poznan", destC/"PL",
date/"27.04.2009", time/"09:00") .

Then, the first query (actually evaluated against the travel database) binds the additional variable borderStation, depending on the values of destC.

In our case, there are three border stations known for traveling to Poland. Thus, three tuples are generated, namely

from	fromC	to	toC	borderStation	date	time
Freiburg	D	Poznan	PL	Szczecin	27.4.2009	09:00
Freiburg	D	Poznan	PL	Frankfurt(Oder)	27.4.2009	09:00
Freiburg	D	Poznan	PL	Görlitz	27.4.2009	09:00

With these tuples, the first DWQL query is evaluated. The tuples are projected and renamed according to the dwql:{input|output}Variable specifications (borderStation is used as dest) and the view germanRailwaysByDept is retrieved for the input tuples

{ (start/"Freiburg", dest/"Szczecin", date/"...", desiredDeptTime/"09:00"),
(start/"Freiburg", dest/"Frankfurt(Oder)", date/"...", desiredDeptTime/"09:00"),
(start/"Freiburg", dest/"Görlitz", date/"...", desiredDeptTime/"09:00") }

returning the following answer tuples:

{ (start/"Freiburg", dest/"Szczecin", date/"...", desiredDeptTime/"09:00",
deptTime/"09:49", arrTime/"18:48", duration/"8:59", price/"131.20"),
:
(start/"Freiburg", dest/"Frankfurt(Oder)", date/"...", desiredDeptTime/"09:00",
deptTime/"09:49", arrTime/"17:26", duration/"7:37", price/"127.00"),
(start/"Freiburg", dest/"Frankfurt(Oder)", date/"...", desiredDeptTime/"09:00",
deptTime/"09:49", arrTime/"17:30", duration/"7:41", price/"131.00"),

```
<ccs:Sequence xmlns:ccs="http://.../languages/2006/ccs#">
  assume variables start, startC, dest, destC, date, and time bound to initial values
  <ccsns:Query>
    binds variable borderStation by query hasBorderStation(startC, destC, borderStation)
  </ccsns:Query>
  <ccs:Query>
    <dwql:Query xmlns:dwql="http://.../languages/2008/dwql#">
      <dwql:view dwql:resource="bla://dwql-views/travel/germanRailwaysByDept" />
      <dwql:inputVariable dwql:name="start" dwql:use="start" />
      <dwql:inputVariable dwql:name="borderStation" dwql:use="dest" />
      <dwql:inputVariable dwql:name="date" dwql:use="date" />
      <dwql:inputVariable dwql:name="time" dwql:use="desiredDeptTime" />
      <dwql:outputVariable dwql:name="arrBorderTime" dwql:use="arrTime" />
      <dwql:outputVariable dwql:name="P1" dwql:use="price" />
    </dwql:Query>
  </ccs:Query>
  <ccs:Alternative>
    <ccs:Sequence>
      <ccs:Test> <ccs:Equals ccs:variable="destC" ccs:withValue="PL" /></ccs:Test>
      <ccs:Query>
        <dwql:Query xmlns:dwql="http://.../languages/2008/dwql#">
          <dwql:view dwql:resource="bla://dwql-views/travel/polishRailwaysByDept" />
          <dwql:inputVariable dwql:name="borderStation" dwql:use="start" />
          <dwql:inputVariable dwql:name="dest" dwql:use="dest" />
          <dwql:inputVariable dwql:name="date" dwql:use="date" />
          <dwql:inputVariable dwql:name="arrBorderTime" dwql:use="desiredDeptTime" />
          <dwql:outputVariable dwql:name="arrTime" dwql:use="arrTime" />
          <dwql:outputVariable dwql:name="P2" dwql:use="price" />
        </dwql:Query>
      </ccs:Query>
      calculate Price := P1 + P2
    </ccs:Sequence>
    similar <ccs:Sequence> specifications for other destination countries
  </ccs:Alternative>
</ccs:Sequence>
```

Fig. 3. Railway Connection Search as a CCS Sequence in XML Markup

:
(start/"Freiburg", dest/"Görlitz", date/"...", desiredDeptTime/"09:00",
deptTime/"10:57", arrTime/"19:27", duration/"8:30", price/"127.00"),
:
} .

Note that the result is just a set of tuples, not a set of groups of tuples and although the underlying interface does not support a set-oriented query interface, DWQL provides a set-oriented interface and iterates internally.

The tuples are then unrenamed (dest → borderStation (for joining), and new arrTime → arrBorderTime and price → P1). Then, the workflow enters the appropriate alternative for querying the railway company in the destination country. The Polish Railways page is wrapped to the same signature. For the input to query, the renaming is borderStation → start and arrBorderTime → desiredDeptTime. The query returns for each tuple the connecting trains from the respective border station to Poznan. The resulting tuples, amongst them

(start/"Frankfurt(Oder)", dest/"Poznan", date/"...", desiredDeptTime/"17:26",
deptTime/"17:33", arrTime/"19:27", duration/"1:54", price/"22.00")

are then unrenamed (start → borderStation, desiredDeptTime → arrBorderTime and price → P2) and joined with the before tuples (where the values of borderStation and arrBorderTime are the actual join condition). Finally, Price is obtained as P1 + P2, considering the different currencies as described below.

```
@prefix : <http://www.semwebtech.org/mars#> .
    ## further prefixes as in Figure 2
[ a :Process;
  useVariables _:start, _:startC, _:dest, _:destC,
       _:date, _:time, _:border, _:arrBT _:p1, _:p2, _:pr ].
_:start  a :Variable; :name "start"; :datatype xsd:string.
_:startC a :Variable; :name "start"; ## ... derived from the first query
_:dest   a :Variable; :name "dest"; :datatype xsd:string.
_:destC  a :Variable; :name "start"; ## ... derived from the first query
_:date   a :Variable; :name "date"; :datatype xsd:date;
         :format "dd.MM.yyyy".
_:border a :Variable; :name "borderStation"; :datatype xsd:string.
_:time   a :Variable; :name "time"; :datatype xsd:time; :format "HH:mm".
_:arrBT  a :Variable; :name "arrBorderTime"; :datatype xsd:time;
         :format "HH:mm".
_:arrT a :Variable; :name "arrTime"; :datatype xsd:time; :format "HH:mm".
_:p1  a :Variable; :name "P1"; :datatype xsd:decimal;
                           :dimension dim:price; :unit curr:EUR.
_:p2  a :Variable; :name "P2"; :datatype xsd:decimal;
                           :dimension dim:price; :unit curr:PLN.
_:pr  a :Variable; :name "price"; :datatype xsd:decimal;
                           :dimension dim:price; :unit curr:EUR.
```

Fig. 4. MARS Knowledge about the Railway Connection Process

5.4 Reasoning about Process Variables

As discussed in Section 3, variables in a MARS workflow are typed, including information about measurements and units. In the above example, the prices are typed and the required date formats are managed.

The MARS knowledge about the process is shown in Figure 4. It is derived completely from the process structure and the DWQL Source Descriptions. The derivation of the variables' properties is similar to *static typing* in programming languages. While most of the properties are straightforward, the prices deserve attention: P1 which is the answer from German Railways, is known to have the unit curr:EUR while P2 which is the answer from Polish Railways, has the unit curr:PLN. Price, which is derived as the sum of P1 + P2 gets also the unit curr:EUR. When computing Price := P1 + P2, for the above sample connection, P2 (e.g., 22 PLN) will be converted in 4.87 EUR before being added.

Additionally, some verification of the workflow's correctess (e.g., correct use of dimension-compatible answers) can be done based on the source annotations.

6 Related Work

Our work is related to the field of Semantic Web Services [4,2,8]. There, Web service descriptions are enhanced with semantic annotations mainly to facilitate automatic service composition [12].

In [1] an approach for annotating Web services is presented that allows to specify propositional and temporal constraints additionally to the regular input and output signature of a Web service. The constraints considered in their work are mandated by side-effects of the invocation of Web services, which is also the case for the Semantic Web service description proposals. Since our Deep Web sources are solely used for collecting information, these issues do not arise in our scenario. [3] presents a method for deriving query access plans for Deep Web sources. They describe the data sources as Datalog predicates with input and output characteristics, ranging over domain classes (i.e. movies). In our approach, we have a more detailed notion of domains, ranging from complex measurements with different syntactical representations to the possibility to use concepts of domain ontologies.

Finally, our work could benefit from complementary work on the analysis of query capabilities for deriving the data types and ranges of input arguments automatically [18,19].

7 Conclusion

For combining Deep Web data in a non-trivial way it is mandatory to assign a precise semantics to the input and output signature of the underlying source. We introduced a comprehensive formalism for annotating Deep Web sources semantically and showed how it is used for composition of different Deep Web services into a query workflow.

A prototype of the MARS framework can be found with sample processes and further documentation at http://www.semwebtech.org/mars/frontend/.

References

1. Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web Service interfaces. In: WWW, pp. 148–159. ACM, New York (2005)
2. Burstein, M.H., Hobbs, J.R., Lassila, O., Martin, D.L., McDermott, D.V., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T.R., Sycara, K.P.: DAML-S: Web Service description for the Semantic Web. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 348–363. Springer, Heidelberg (2002)
3. Calì, A., Martinenghi, D.: Querying Data under Access Limitations. In: ICDE, pp. 50–59. IEEE, Los Alamitos (2008)
4. de Bruijn, J., Lausen, H., Polleres, A., Fensel, D.: The Web Service modeling language WSML: An overview. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 590–604. Springer, Heidelberg (2006)
5. Fritzen, O., May, W., Schenk, F.: Markup and Component Interoperability for Active Rules. In: Calvanese, D., Lausen, G. (eds.) RR 2008. LNCS, vol. 5341, pp. 197–204. Springer, Heidelberg (2008)
6. Hornung, T., May, W., Lausen, G.: Process algebra-based query workflows. In: CAiSE (to appear, 2009)
7. Hornung, T., Simon, K., Lausen, G.: Mashups over the Deep Web. In: WEBIST 2008. LNBIP, vol. 18, pp. 228–241. Springer, Heidelberg (2009)
8. Martin, D., Paolucci, M., Wagner, M.: Bringing semantic annotations to web services: OWL-S from the SAWSDL perspective. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 340–352. Springer, Heidelberg (2007)
9. May, W., Alferes, J.J., Amador, R.: Active rules in the Semantic Web: Dealing with language heterogeneity. In: Adi, A., Stoutenburg, S., Tabet, S. (eds.) RuleML 2005. LNCS, vol. 3791, pp. 30–44. Springer, Heidelberg (2005)
10. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science, pp. 267–310 (1983)
11. OWL Web Ontology Language (2004), http://www.w3.org/TR/owl-features/
12. Rao, J., Su, X.: A survey of automated Web Service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2004)
13. Resource Description Framework (RDF) (2000), http://www.w3.org/RDF
14. Simon, K., Lausen, G.: Viper: Augmenting automatic information extraction with visual perceptions. In: CIKM, pp. 381–388. ACM, New York (2005)
15. SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/
16. Wang, Y., Hornung, T.: Deep Web Navigation by Example. In: BIS (Workshops), CEUR Workshop Proceedings 333, pp. 131–140. CEUR-WS.org (2008)
17. Turtle - Terse RDF Triple Language, http://www.dajobe.org/2004/01/turtle/
18. Wu, W., Yu, C.T., Doan, A., Meng, W.: An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In: SIGMOD, pp. 95–106 (2004)
19. Zhang, Z., He, B., Chang, K.C.-C.: Understanding Web query interfaces: Best-effort parsing with hidden syntax. In: SIGMOD, pp. 107–118 (2004)