XML Querying Using Ontological Information

Hans Eric Svensson and Artur Wilk

Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
{x05erisv, artwi}@ida.liu.se

Abstract. The paper addresses the problem of using semantic annotations in XML documents for better querying XML data. We assume that the annotations refer to an ontology defined in OWL (Web Ontology Language). The intention is then to combine syntactic querying techniques on XML documents with OWL ontology reasoning to filter out semantically irrelevant answers. The solution presented in this paper is an extension of the declarative rule-based XML query and transformation language Xcerpt. The extension allows to interface an ontology reasoner from Xcerpt rules. This makes it possible to use Xcerpt to filter extracted XML data using ontological information. Additionally it allows to retrieve ontological information by sending semantic queries to a reasoner. The prototype implementation uses DIG (Description Logic interface) for communication with the OWL reasoner RacerPro where the ontology queries are answered.

1 Introduction

XML, designed by W3C[1], is increasingly used for representing semistructured data on the Web. XML is considered a basic layer in the W3C Semantic Web initiative initiated by Tim Berners-Lee. As stated by Antoniou and van Harmelen [2] the objective of the initiative is "to represent Web content in a form that is more easily machine-processable and to use intelligent techniques to take advantage of these representations". The intention is not to build a new Web from scratch, but to stimulate gradual evolvement of the existing Web in the above-mentioned direction.

Another layer of the Semantic Web is the so-called ontology layer. Ontologies provide information about concepts, roles and individuals in a given application domain. Thus an ontology gives a common vocabulary to be understood in the same way by various applications in the domain. For example the concept *tree* in graph theory applications is understood to be a special kind of the concept *graph*. The same concept would be understood to be a special kind of the concept *plant* in a botanical vocabulary. The roles defined by an ontology are binary relations on concepts. The Web Ontology Language OWL [1], recommended by the W3C, is used for specifying Web ontologies. Formally the language is based on a Description Logic. An OWL ontology can thus be seen as a set of logical

[1] http://www.w3.org/

axioms. Querying a given ontology is done by reasoning in the underlying logic. For example, if the graph ontology states that the individual t is a *tree* it can be concluded that t is a *graph*.

XML is supported by query languages, including the W3C Candidate Recommendation XQuery [4]. Querying of XML data in such languages relies on the structure of the queried XML data: a query identifies a (possibly empty) set of fragments of given XML data. The structure-based querying of XML data is thus based on the syntax of the data. XML data may include semantic annotations, referring to concepts defined by ontologies. However, XML query languages do not provide ontology reasoning capabilities. The objective of this paper is to show how structure-based querying can be combined with ontology reasoning. For this we combine the XML query language Xcerpt [7,6] with ontology queries. Xcerpt is being developed by the EU Network of Excellence REWERSE [2] in the 6th Framework Programme. It differs from most other XML query languages in that it is deductive and rule based. This makes it more suitable for integration with ontology queries.

As already stated, the objective of our work is to enhance structural querying of XML data with ontology reasoning. We assume that XML data contains annotations referring to an ontology defined in OWL. We would like to filter XML documents returned by a structural query by reasoning on semantic annotations included therein. This can be illustrated by the following example. Assume that an XML database of culinary recipes is given. Each recipe indicates ingredients (like flour, salt, sugar etc.). We assume that the names of the ingredients are defined by a standard ontology, accessible separately on the web and providing also some classification. For example, the standard may specify disjoint classes of gluten-containing and gluten-free ingredients (see Figure 1). Thus, the names of ingredients in the XML recipe can be seen as semantic annotations. To prepare dinner we would query the XML database for recipes. To check if the ingredients of a chosen recipe are gluten-free we have additionally to query the ontology.

Thus, the problem outlined above can be seen as the problem of interfacing of an XML query language with an ontology reasoner. We decided to choose Xcerpt as the query language as variables of Xcerpt can naturally be used for passing semantic annotations from results of Xcerpt queries to an ontology reasoner. Also we had access to the source code of the Xcerpt implementation which made it possible to implement our solution by modification of this code. The prototype implementation uses DIG (Description Logic interface [3]) for communication with the OWL reasoner RacerPro[3] where the ontology queries are answered.

The prototype implements two ways of interfacing a reasoner from Xcerpt. One of them involves boolean ontology queries, which are used to filter out irrelevant answers. Another one allows arbitrary DIG queries to retrieve ontological information from the reasoner. Such information can be further used by other rules in an Xcerpt program.

---

[2] `http://www.rewerse.net/`

[3] `http://www.racer-systems.com/`

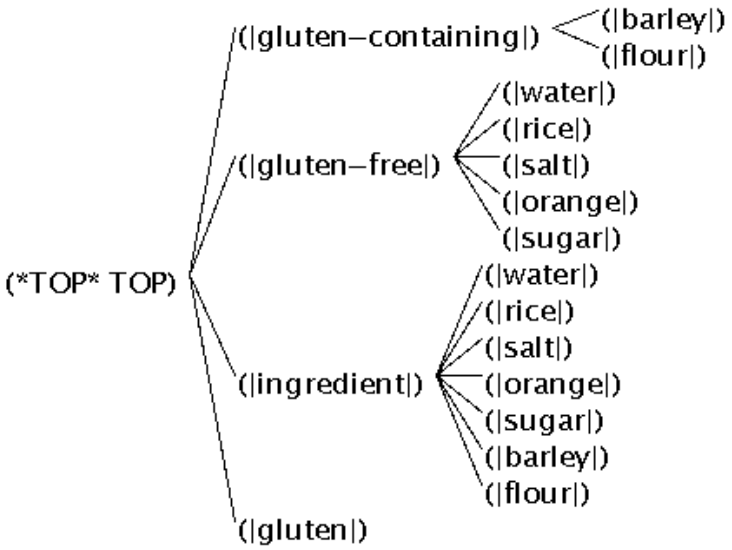

Fig. 1. Recipe ontology graph (generated by RacerPorter - a graphical user interface of RacerPro).

The rest of the paper is organised as follows. Section 2 briefly introduces the query language Xcerpt and gives some background information on the DIG interface. Section 3 presents an extension of Xcerpt allowing querying XML using ontological information. It also presents a prototype implementing new constructs in Xcerpt. Finally, Section 4 provides some conclusions.

2 Preliminaries

This section gives a brief introduction to the XML query and transformation language Xcerpt and the DIG Interface. These are basic techniques applied in the presented work.

2.1 Xcerpt

An Xcerpt program is a set of rules consisting of a body and of a head. The body of a rule is a query intended to match data terms. If the query contains variables such matching results in answer substitutions for variables. The head uses the results of matching to construct new data terms. The queried data is either specified in the body or is produced by rules of the program. There are two kinds of rules: goal rules produce the final output of the program, while construct

rules produce intermediate data, which can be further queried by other rules. Their syntax is as follows:

```
GOAL          CONSTRUCT
  head          head
FROM          FROM
  body          body
END           END
```

Sometimes, we will denote the rules as $head \leftarrow body$ neglecting distinction between goal and construct rules.

XML data is represented in Xcerpt as **data terms**. Data terms are built from basic constants and labels using two kinds of parentheses: brackets [ ] and braces { }. Basic constants represent basic values such as attribute values and character data (called PCDATA). A label represents an XML element name. The parentheses following a label include a sequence of data terms (its direct subterms). Brackets are used to indicate that the direct subterms are ordered (in the order of their occurrence in the sequence), while braces indicate that the direct subterms are unordered. The latter alternative is used to encode attributes of an XML element by a data term of the form $attr\{l_1[v_1], \ldots, l_n[v_n]\}$ where l_i are names of the attributes and v_i are their respective values.

Example 1. This is an XML element and the corresponding data term.

```
<CD price="9.90">            CD[ attr{ price["9.90"] },
  <title>Empire</title>          title[ "Empire" ],
  <artist>Bob Dylan</artist>     artist["Bob Dylan"],
  <country>USA</country>         country["USA"]
</CD>                        ]
```

There are two other kinds of terms in Xcerpt: query terms and construct terms.

Query terms are (possibly incomplete) patterns which are used in a rule body (query) to match data terms. In particular, every data term is a query term. Generally query terms may include variables so that a successful matching binds variables of a query term to data terms. Such bindings are called answer substitutions. A result of a query term matching a data term is a set of answer substitutions. For example a query term $a[\, b[\,], \mathtt{var}\, X\,]$ matches a data term $a[\, b[\,], c[\,]\,]$ resulting in answer substitution set $\{X/c[\,]\}$. Query terms can be ordered or unordered patterns, denoted, respectively, by brackets and braces. For example a query term $a[\, c[\,], b[\,]\,]$ is an ordered pattern and it does not match a data term $a[\, b[\,], c[\,]\,]$ but a query term $a\{\, c[\,], b[\,]\,\}$, which is an unordered pattern, matches $a[\, b[\,], c[\,]\,]$. Query terms with double brackets or braces are incomplete patterns. For example a query term $a[[\, b[\,], d[\,]\,]]$ is an incomplete pattern which matches a data term $a[\, b[\,], c[\,], d[\,]\,]$. As the query term uses brackets the matching subterms of the data term must occur in the same order as in the pattern. Thus the query term $a[[\, b[\,], d[\,]\,]]$ does not match a data term $a[\, d[\,], b[\,], c[\,]\,]$. In

contrast a query term $a\{\{\, b[\,], d[\,]\,\}\}$ matches $a[\, d[\,], b[\,], c[\,]\,]$. To specify subterms at arbitrary depth a keyword `desc` is used e.g. a query term `desc` $d[\,]$ matches a data term $a[\, b[\, d[\,]\,], c[\,]\,]$.

A query term q in a rule body may be associated with a resource r storing XML data or data terms. This is done by a construction of the form `in`$[r, q]$. The meaning of this construction is that q is to be matched against data in r. Query terms in the body of a rule which have no associated resource are matched against data generated by rules of the Xcerpt program.

Rule bodies are constructed from query terms (possibly with indicated resources) using logical connectives such as `or`, `and`, and `not`.

Construct terms are used in rule heads to construct new data terms. They are similar to data terms, but may contain variables which act as place holders for data selected in a query. They may also use a grouping construct `all` which is used to collect all instances that result from different variable bindings [5].

A construct term c in a goal rule head may be associated with a resource r to which the goal results are written. This is done by a construction of the form `out`$[r, c]$. If a head of a goal rule is a construct term which is not associated with a resource the results of the rule are directed to the standard output.

Example 2. Consider a document *catalogue.xcerpt* containing a data term:

```
catalogue[
  cd[ title["Empire"], artist["Bob Dylan"], year["1985"] ],
  cd[ title["Hide your heart"], artist["Bonnie Tyler"], year["1988"] ],
  cd[ title["Stop"], artist["Sam Brown"], year[ "1988"] ]
]
```

Here is an Xcerpt rule which queries the document and extracts titles and artists of the CD's issued in 1988 and presents the results in a changed form (title as name and artist as author).

```
GOAL
  results [
    all result[ name[ var TITLE ], author[ var ARTIST ] ]
]
FROM
  in[ "file:catalogue.xcerpt",
    catalogue{{
      cd{ title[ var TITLE ], artist[ var ARTIST ], year[ "1988" ] }
    }}
  ]
END
```

The results returned by the rule are:

```
results[ result[ name[ "Hide your heart" ], author[ "Bonnie Tyler" ] ],
         result[ name[ "Stop" ], author[ "Sam Brown" ] ] ]
```

Xcerpt rules may be chained to form complex query programs, i.e. rules may query the results of other rules.

2.2 DIG interface

Ontologies provide information about concepts, roles and individuals in a given application domain. Thus an ontology gives a common vocabulary to be understood in the same way by various applications in the domain. A main language used to defined ontologies is OWL developed by W3C. OWL is based on description logics.

An OWL file representing an ontology is just an encoding of a set of axioms. To make use of the axioms one needs an ontology reasoner. Using an ontology reasoner it is possible to draw conclusions from the set of axioms such as discovering implicit subclass relationships and discovering class equivalence. In the presented work we use the ontology reasoner RacerPro. To allow Xcerpt programs to communicate with the reasoner we need to use a reasoner interface. For this purpose we have chosen DIG (Description Logic interface [3]) which is supported by RacerPro.

The DIG interface is an API for a general description logic system. It is capable of expressing class and property expressions common to most description logics. Using DIG clients can communicate with a reasoner through the use of HTTP POST requests. The request is an XML encoded message of one of the following types: management, ask or tell. Management requests are used e.g. to identify the reasoner along with its capabilities or to allocate a new knowledge base and return its unique identifier. Tell requests, expressed in the *Tell* language, are used to make assertions into the reasoner's knowledge base. Ask requests, expressed in the *Ask* language, are used to query the knowledge base. Replies to ask requests are provided with the *Response* language. Tell, Ask and Response languages use expressions from the *Concept* language which is used to define classes, properties, declare individuals etc. Here we present an extract of expressions from the Concept language:

- Primitive concepts, roles and individuals:
  - `<top/>` - the universal concept (like owl:Thing)
  - `<bottom/>` - the empty concept (like owl:Nothing)
  - `<catom name="`CN`"/>` - introduces a concept (i.e. class) CN
  - `<ratom val="`RN`"/>` - introduces a role (i.e. property) RN
  - `<individual name="`IN`"/>` - introduces an individual IN
- Boolean operators:
  - `<and>`$C_1 \ldots C_n$`</and>` - intersection of concept expressions $C_1, \ldots, C_n$
  - `<or>`$C_1 \ldots C_n$`</or>` - union of concept expressions $C_1, \ldots, C_n$
  - `<not>`C`</not>` - complement of a concept expression C

This is an excerpt from the Ask language ($C, C_1, \ldots, C_n$ are concept expressions):

- satisfiability queries for which the response is a boolean value
  - `<satisfiable>`C`</satisfiable>`
  - `<subsumes>`$C_1\, C_2$`</subsumes>`
  - `<disjoint>`$C_1\, C_2$`</disjoint>`

– concept retrieval queries for which the response is a set of concepts
  - `<allConceptNames/>`
  - `<parents>`C`</parents>`
  - `<children>`C`</children>`
  - `<descendants>`C`</descendants>`
  - `<equivalents>`C`</equivalents>`

3 Extended Xcerpt

This section presents a way for extending Xcerpt to enable it to interface with an ontology reasoner while querying XML data. First we present a kind of filter used in Xcerpt rules to filter out semantically irrelevant answers. Then, we propose a more general way for interfacing an ontology reasoner with Xcerpt.

In order not to confuse keywords from Xcerpt like `or`, `and` etc., with similar keywords used in DIG, we precede them with the character '!'. Thus, for example, `!or` in Extended Xcerpt is equivalent to `or` in Xcerpt. Also, the character '!' is used to denote a label of a data term representing attributes (!*attr*).

3.1 Answer filtering

Here we present a new Xcerpt construction called *filter*. Such a filter can be used between a body and a head of an Extended Xcerpt rule to filter out semantically irrelevant answers:

`GOAL`	`CONSTRUCT`
head	*head*
`FILTER`	`FILTER`
filter	*filter*
`FROM`	`FROM`
body	*body*
`END`	`END`

A *filter* is an expression `!dig`[U_{RL}, *cterm*], where U_{RL} is an URL of an ontology reasoner answering DIG queries and *cterm* is a construct term used to produce a DIG query. Evaluation of a body of a rule results in a set Ψ of answer substitutions. The substitutions are then used by the construct term *cterm* to build a data term $asks[a_1, \ldots, a_n]$ where $a_1, \ldots, a_n$ are expressions from the DIG's Ask language for which boolean answers can be given. The data term $asks[a_1, \ldots, a_n]$ corresponds to a result of an Xcerpt rule $cterm \leftarrow body$. The data term is transformed into an XML document and sent to an ontology reasoner specified by U_{RL}. The XML document sent to the reasoner additionally contains a header with DIG namespace declarations and unique identifiers for the elements corresponding to $a_1, \ldots, a_n$. The reasoner replies with a boolean answer for each ask expression. If the answer for the query a_i is 'false' the answer substitutions used to construct a_i are discarded[4]; otherwise they are retained.

[4] As *cterm* may contain grouping constructs a_i may originate from more than one answer substitution.

As a result we obtain a subset Ψ' of the set of answer substitutions Ψ. Only the substitutions from Ψ' are used then to build the results of the initial rule.

Our prototype of Extended Xcerpt implements this method of filtering. However, the present version of the prototype is somewhat restricted. It only allows filters in goal rules and forbids use of grouping constructs such as `!all` in filters. As the grouping constructs are forbidden there is no need to use explicitly a common label *asks* for ask expressions in the filter. Thus a construct term *cterm* in a filter `!dig`[U_{RL}, *cterm*] is used to build separate ask expressions a_i for each answer substitution from Ψ. Then a data term $asks[a_1, \ldots, a_n]$ is built automatically; it is translated into XML and sent to a reasoner. In order to be able to further query the results of goal rules with filters the Xcerpt implementation has been altered in such way that the files produced by goal rules can be queried by other goal rules. The goal rules are evaluated in the order they appear in a program.

Usage of the filter is illustrated on the following example which can be run on the prototype. Consider an XML document *recipes.xml*, which is a collection of culinary recipes. The document is represented by the data term:

```
recipes[
 recipe[
   name[ "Recipe1" ],
   ingredients[
     ingr[ name[ "sugar" ], amount [ !attr{ unit[ "tbsp" ] }, 3 ] ],
     ingr[ name[ "orange" ], amount[ !attr{ unit[ "unit" ] }, 1 ] ]
   ]
 ]
 recipe[
   name[ "Recipe2" ],
   ingredients[
     ingr[ name[ "flour" ], amount[ !attr{ unit[ "dl" ] }, 3 ] ],
     ingr[ name[ "salt" ], amount[ !attr{ unit[ "krm" ] }, 1 ] ]
   ]
 ]
 recipe[
   name [ "Recipe3" ],
   ingredients[
     ingr[ name[ "barley" ], amount[ !attr{ unit[ "dl" ] }, 1 ] ],
     ingr[ name[ "salt" ], amount[ !attr{ unit[ "dl" ] }, 2 ] ]
   ]
 ]
]
```

Also consider the culinary ingredients ontology from introduction (Figure 1). We assume that the ontology is loaded into an ontology reasoner which is accessible via the URL `http://localhost:14159/`. We also assume that the names of the ingredients used in the XML document are defined by the ontology. Thus, ingredients in the XML recipe can be seen as semantic annotations. We want to find all the recipes in the XML document which are gluten-free. This can be

achieved using the following program with two goal rules, one of them with a filter:

```
GOAL
  !out [
    !resource [ "file:bad-recipes.xcerpt" ],
    bad-recipe-names [ !all name [ var R ] ] ]
FILTER
  !dig [ "http://localhost:14159/",
      subsumes [
        catom [ !attr { name [ "gluten-containing" ] } ],
        catom [ !attr { name [ var N ] } ] ] ]
FROM
  !in [ !resource [ "file:recipes.xml" ],
    recipes [[
      recipe [[
        name [ var R ],
        ingredients [[ ingredient [[ name [ var N ] ]] ]] ]] ]] ]
END

GOAL
    recipes [ !all name [ var R ] ]
FROM
  and[
    !in [ !resource [ "file:recipes.xml"],
      recipes [[ recipe [[ name [ var R ] ]] ]] ],
    !in [ !resource [ "file:bad-recipes.xcerpt"],
      not bad-recipe-names [[ name [ var R ] ]] ]
  ]
END
```

Evaluation of the program starts from the first goal rule. Evaluation of the body of the rule results in the set of answer substitutions $\Psi = \{\{R/"Recipe1", N/"sugar"\}, \{R/"Recipe1", N/"orange"\}, \{R/"Recipe2", N/"flour"\}, \{R/"Recipe2", N/"salt"\}, \{R/"Recipe3", N/"barley"\}, \{R/"Recipe3", N/"salt"\}\}$. Then the substitution set Ψ is used in construct term from the filter to build data terms representing ask expressions. A separate ask expression is built for each substitution from Ψ. The obtained ask expressions are grouped together under a common label *asks*:

```
asks[
  subsumes[
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "sugar" ] } ] ],
  subsumes[
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "orange" ] } ] ],
  subsumes[
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "flour" ] } ] ],
  subsumes[
```

```
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "salt" ] } ] ],
  subsumes[
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "barley" ] } ] ],
  subsumes[
    catom[ attr{ name [ "gluten-containing" ] } ],
    catom[ attr{ name [ "salt" ] } ] ] ]
```

The *asks* data term is sent to a reasoner. The reasoner replies with a positive answer for the third and the fifth ask expressions as only flour and barley contain gluten wrt. the ontology. As the answers for the remaining ask expressions are negative the substitutions used to build them are discarded. Thus, the obtained set of substitutions used to build the final result of the rule is $\Psi' = \{\{R/"Recipe2", N/"flour"\}, \{R/"Recipe3", N/"barley"\}\}$. Hence, the final result written by the first goal rule into the file *bad-recipes.xcerpt* is:

```
bad-recipe-names[ name["Recipe 2"], name ["Recipe 3"] ]
```

The second goal rule returns those names of recipes from *recipes.xml* which are not in *bad-recipes.xcerpt*. Thus the final result of the program returned by the second goal rule is

```
recipes[ name["Recipe 1"] ]
```

The kind of queries which can be sent to a reasoner is limited due to the DIG interface which is often not sufficiently expressive. It lacks e.g. logical operators such as *and* and *or* (keywords *and* and *or* are used in DIG to denote intersection and union of concepts, respectively). This is a reason why we had to use two goal rules instead of one rule in the example above. To obtain the same result using a program with only one rule we need to be able to use grouping constructs in a filter and e.g. conjunction in the ask expression constructed by the filter. The latter would be needed to assure that each ingredient of a recipe is subsumed by the concept *gluten-free*.

3.2 DIG rules - querying ontology reasoner with Xcerpt

In the previous section we introduced a filter which sends boolean queries to an ontology reasoner and based on the reasoner replies, filters out irrelevant answers. However, we can take more general approach where the queries sent to a reasoner are arbitrary DIG ask expressions (not only boolean). An ordinary Xcerpt rule, say ask rule, can be used to produce such an ask expression which is sent to a reasoner. Then another Xcerpt rule, say response rule, captures the response received from the reasoner and transforms it to a desired format.

This can be reflected by a higher level rule called e.g. a DIG rule. A DIG rule can be denoted as $(h_R \leftarrow b_R) \leftarrowtail (h_A \leftarrow b_A)$, where $h_A \leftarrow b_A$ is an ask rule and $h_R \leftarrow b_R$ is a response rule. Thus h_A is a construct term of the form $asks[\ldots]$ and b_R a query term of the form e.g. $responses\{\{\ldots\}\}$. DIG rules could be handled by an external application which executes relevant Xcerpt programs

and communicates with a reasoner. Another solution is extending Xcerpt itself with DIG rules so it interfaces an ontology reasoner. Beside ordinary rules (i.e. Xcerpt query rules) such an extended Xcerpt could use response and ask rules, respectively, of the forms:

```
CONSTRUCT                    CONSTRUCT
  h_R                          !out[!dig[ U_RL], h_A ]
FROM                         FROM
  !in[ !dig[ U_RL], b_R ]      b_A
END                          END
```

To implement this idea based on backward rule chaining we need to assure that a response rule invokes a relevant ask rule, the result of the ask rule is sent to a reasoner, and the reasoner response is queried by the initial response rule.

We can consider a special, simple case of a DIG rule $(h_R \leftarrow b_R) \leftarrowtail (h_A \leftarrow b_A)$, where its body, the ask rule, is of the form $h_A \leftarrow$ `!and[ ]`. Thus the ask rule is equivalent to a data term h_A which represents fixed ask expressions i.e. h_A is a data term $asks[\ldots]$. Such a simple DIG rule can be denoted as $(h_R \leftarrow b_R) \leftarrowtail h_A$. The prototype of Extended Xcerpt is restricted to such simple DIG rules. DIG rules $(h_R \leftarrow b_R) \leftarrowtail h_A$ are incorporated into Xcerpt goal rules which are of the form:

```
GOAL
  h_R
FROM
  !in[ !dig[ U_RL, h_A ], b_R ]
END
```

h_A is a data term $asks[a_1, \ldots, a_n]$ containing ask expressions $a_1, \ldots, a_n$ or a data term $tells[t_1, \ldots, t_n]$ containing tell expressions $t_1, \ldots, t_n$. Alternatively, h_A can be a URI of an XML file storing an ask or tell expression. The ask expressions $a_1, \ldots, a_n$ (and tell expressions) must contain unique identifiers to be able to relate reasoner responses with them. As the programmer handles the reasoner responses by himself/herself this time the identifiers cannot be added automatically. The rule is evaluated in the following way. The data term h_A is transformed into an XML document to which a header containing DIG namespace declarations is added. Such a document is sent to the reasoner specified by U_{RL}. The response returned by the reasoner is queried by the query b_R. Then the resulting answer substitutions are applied to a construct term h_R and a rule result is returned.

Consider the following example. We want to query the ingredients ontology to build a document containing gluten-free ingredients. We use the following rule:

```
GOAL
  results [ !all var C ]
FROM
 !dig [ "http://localhost:14159/",
    asks [
```

```
      descendants [
        !attr{ id [ "q1" ] },
        catom [[ !attr { name [ "gluten-free" ]} ]],
      ]
    ],
    responses {{
        conceptSet {{
          !attr { id [ "q1" ] },
          synonyms [[
            catom [[ !attr { name [ var C ] } ]]
          ]]
        }}
    }}
  ]
```

The result returned by the rule is

```
results [
   "water",
   "rice",
   "salt",
   "orange",
   "sugar"
 ]
```

Although the approach using DIG rules is in some sense more general than answer filtering presented in the previous section, it cannot be used directly for answer filtering. This is because a response rule can only query a response of the reasoner and does not have access to the answers of the body of the ask rule. Thus the answers cannot be filtered based on the reasoner responses. However, a workaround for achieving the same goal as with answer filtering is possible. First, the needed ontological information could be captured by a DIG rule. Then an ordinary Xcerpt rule could query both an XML document and the ontological data obtained from a reasoner. In this way the irrelevant (wrt. the ontology) XML data could be filtered out.

4 Conclusions

The paper addresses the problem of how to use ontological information in the context of querying XML data. The problem seems to be important for achieving the Semantic Web but it is not sufficiently covered in literature. The solution proposed in this paper extends the XML query language Xcerpt by allowing the combination of XML queries with ontology queries. The extension allows Xcerpt rules to communicate with an ontology reasoner using the DIG interface. We presented two ways of extension. The first of them is a kind of a filter used in between the body and the head of an Xcerpt rule to filter out semantically irrelevant answers. Another approach, which is more general in some sense, allows interfacing an ontology reasoner with arbitrary DIG queries.

A restricted version of the presented techniques of interfacing an ontology reasoner are implemented in a prototype of Extended Xcerpt. The prototype requires further development. Allowing grouping constructs in filters, filters in construct rules and unrestricted DIG rules would substantially increase the functionality of the prototype.

Acknowledgment

We would like to thank Professor Jan Małuszyński for his interesting ideas initiating this work.

This research has been funded by the European Commission and by the Swiss State Secretariat for Education and Research within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

In the work presented here the RacerPro Software was used under a free educational license from Racer Systems GmbH & Co. KG[5] for ontology reasoning.

References

1. OWL Web Ontology Language Overview. February 2004. W3C Recommendation. `http://www.w3.org/TR/owl-features/`.
2. G. Antoniou and F.van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.
3. S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. In *Proceedings of DL2003 Workshop*, Rome, 2003.
4. W3 Consortium. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/2005/WD-xquery-20050915/`.
5. T. Furche, F. Bry, and O. Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML in Web Querying. In *International Workshop, PSWR 2005, Dagstuhl Castle, Germany, September 2005, Proceedings*, number 3703 in LNCS, pages 72–84. Springer Verlag, 2005.
6. S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, Germany, 2004.
7. S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

[5] `http://www.racer-systems.com/`