Combining XML querying with ontology reasoning: Xcerpt and DIG

Włodzimierz Drabent[1,2], Artur Wilk[1]
[1]Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
[2]Institute of Computer Science, Polish Academy of Sciences,
ul. Ordona 21, Pl – 01-237 Warszawa, Poland
{wlodr, artwi}@ida.liu.se

Abstract

The paper addresses the problem of combining ontological reasoning with querying XML data. We present an extension of a rule-based XML query and transformation language Xcerpt. The extension allows to interface an ontology reasoner from Xcerpt programs. In this way querying can employ the ontology information, for instance to filter out semantically irrelevant answers. Communication between Xcerpt programs and ontology reasoner is based on DIG interface. The extension can be implemented without modifying the underlying Xcerpt implementation.

1. Introduction

XML, designed by W3C[1], is increasingly used for representing semistructured data on the Web. XML is supported by query languages, including the W3C Candidate Recommendation XQuery [11]. 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 [9, 8] 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 rule based and uses pattern matching instead of path navigation for locating and extracting data.

[1] `http://www.w3.org/`
[2] `http://www.rewerse.net/`

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. We would like to query XML data using this ontological information. For instance we may want 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 ontology 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 a gluten-free meal we would query the XML database for recipes, and query the ontology to check if the ingredients are gluten-free.

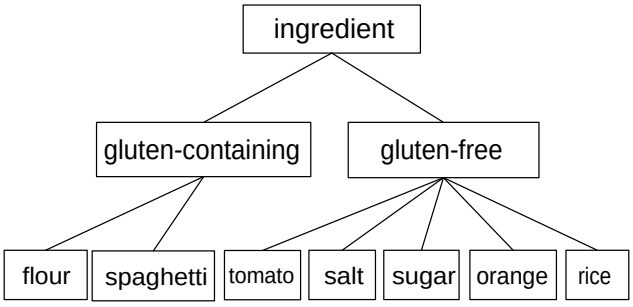

Figure 1. Recipe ontology graph

Thus, the problem outlined above can be seen as the problem of interfacing an XML query language with an ontology reasoner. We propose a solution for XML query language Xcerpt and any ontology reasoner supporting DIG

interface. DIG is a standard interface to ontology reasoners, supported by e.g. RacerPro[3] or Pellet[4].

The presented work is inspired by [1] where a framework for hybrid combination of rule languages with constraint languages is proposed. Combination of (a negation-free subset of) Xcerpt and an ontology language is proposed as a possible instance of the framework. Our approach is different. It imposes no restrictions on Xcerpt. It allows arbitrary ontology queries expressible in DIG, while [1] considers only Boolean queries. The fixpoint semantics of the approach of [1] is equivalent to performing the rule computation first and then evaluating the ontology queries (that have been accumulated during the rule computation). Our semantics does not have this property.

The presented work continues the efforts described in [10] and introduces substantial improvements. One of them is that the language Extended Xcerpt can be implemented without modification of the underlying Xcerpt implementation. The previous paper presents two methods of interaction between Xcerpt and a reasoner; the approach presented here is more general than each of them. It allows more flexible semantic filtering of data. Also due to a possibility of passing a query context from ask to response rules we can build more interesting queries.

The problem of combination of XML queries with ontology queries seems to be important for the Semantic Web. There exist approaches, like SWRL[5] in which rules and ontologies are combined into one logical language. Usually such a language combines the semantics of ontology and of rules in a sophisticated way. In contrast, we are interested in a hybrid approach where a rule language and an ontology language are interfaced, preserving their original semantics as much as possible. Previous related work addresses mostly integration of logic programming language Datalog with Description Logics. Early work of this kind is represented by AL-log [4] and CARIN [6], where Datalog rule bodies are extended with queries to a given ontology (DL axioms). The authors discuss theoretical foundations as well as implementation of such an integration, but do not consider the context of the Semantic Web. More recently, Eiter et al. [5] extend Datalog with so-called DL-queries which make it possible to query locally modified DL axioms thus allowing information flow in both directions between rules and ontologies. A theoretical study of integration of logic programming rules and DL ontologies is presented by Rosati [7] (see also references therein for other related work). In contrast to the above mentioned papers our rules are not Datalog clauses but Xcerpt queries.

The rest of the paper is organized 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 provides program examples illustrating a usage of the extended system. Finally, Section 4 provides some conclusions.

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

[4] http://www.mindswap.org/2003/pellet/

[5] http://www.w3.org/Submission/SWRL/

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* ← *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">
  <title>Empire</title>
  <artist>Bob Dylan</artist>
</CD>
```

```
CD[
  attr{ price["9.90"] },
  title[ "Empire" ],
  artist[ "Bob Dylan" ]
]
```
□

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 $a[\,b[\,], \texttt{var}\, X\,]$ 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 $\texttt{desc}\, d[\,]$ 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 $\texttt{in}[r, q]$. 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.

Queries are constructed from query terms (possibly with indicated resources) using logical connectives such as `or`, `and`, and `not`. A rule body is a query.

Construct terms are used in rule heads to construct new data terms. They are similar to data terms, but may contain variables. Data terms are constructed out of construct terms by applying answer substitutions obtained from a rule body. Construct terms may also use a grouping construct `all` which is used to collect all instances that result from different variable bindings.

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 result returned by the rule is:

```
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. To communicate with the reasoner we need to use a reasoner interface. For this purpose we have chosen DIG interface [2] which is supported by many reasoners.

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, containing *tell* statements, are used to make assertions into the reasoner's knowledge base. Ask requests, containing *ask* statements, are used to query the knowledge base. Replies to ask requests contain response statements. Tell, ask and response statements are build out of concept statements which are used to denote classes, properties, individuals etc. Here we present an extract of DIG statements used in our examples ($C, C_1, C_2, \ldots$ are concept statements):

- Concept statements:
  - `<catom name="` *CN*`"/>` – a concept (class) *CN*
  - `<ratom val="`*RN*`"/>` – a role (property) *RN*
  - `<some>` *R C* `</some>` – the concept whose objects are in relation R with some objects of a concept C (it corresponds to the formula $\exists R.C$ in description logics)
- Tell statements:
  - `<defconcept name="` *CN*`"/>` – introduces a concept *CN*
  - `<defrole name="` *RN*`"/>` – introduces a role *RN*
  - `<impliesc>`C_1 C_2`</impliesc>` – introduces an axiom stating that a concept C_1 is subsumed by a concept C_2
- Ask statements:
  - `<subsumes>`C_1 C_2`</subsumes>` – Boolean query, asks whether a concept C_2 is subsumed by a concept C_1
  - `<children>`C`</children>` – asks for the list of direct subclasses of a concept C
- Response statements:
  - `<true/>` – if a statement is a logical consequence of the axioms in the knowledge base
  - `<false/>` – if a statement is not a logical consequence of the axioms in the knowledge base
  - `<error/>` – if, for instance, a concept queried about is not defined in the knowledge base
  - `<conceptSet>`
    `<synonyms>` $C_{11} \ldots C_{1n_1}$ `</synonyms>`
    `...`
    `<synonyms>` $C_{m1} \ldots C_{mn_m}$ `</synonyms>`
    `</conceptSet>`

Elements of the Ask and Response language contain also attributes. For instance, the attribute *id* is used to associate the obtained answers with the submitted queries.

Example 3. This is an example of a query request to be sent to an ontology reasoner. It contains three queries. The first two ask whether concepts *sugar* and *potato* are subclasses of the concept *gluten-containing*. The third one asks for direct subclasses of the class *gluten-containing*.

```
<?xml version="1.0"?>
<asks
xmlns="http://dl.kr.org/dig/2003/02/lang">
xmlns:xsi="http://www.w3.org/2001/..."
xsi:schemaLocation="http://dl.kr.org/dig/...
http://dl-web.man.ac.uk/dig/2003/02/dig.xsd"
uri="urn:uri_of_knowledge-base">
  <subsumes id="q1">
    <catom name="gluten-containing"/>
    <catom name="sugar"/>
  </subsumes>
  <subsumes id="q2">
    <catom name="gluten-containing"/>
    <catom name="potato"/>
  </subsumes>
  <children id="q3">
    <catom name="gluten-containing"/>
  </children>
</asks>
```

This is a possible response to the query:

```
<?xml version="1.0"?>
<responses
xmlns="http://dl.kr.org/dig/2003/02/lang">
xmlns:xsi="http://www.w3.org/2001/..."
xsi:schemaLocation="http://dl.kr.org/dig/...
http://dl-web.man.ac.uk/dig/2003/02/dig.xsd">
  <true id="q1"/>
  <error id="q2" message="Undefined concept
     name potato in TBox DEFAULT"
  </error>
  <conceptSet id="q3">
    <synonyms>
      <catom name="flour"/>
    </synonyms>
    <synonyms>
```

```
      <catom name="spaghetti"/>
    </synonyms>
  </conceptSet>
</responses>
```

3. Interfacing ontology reasoner with Xcerpt

The main idea of our extension of Xcerpt is that an Xcerpt rule can be used to produce Ask queries to be delivered to an ontology reasoner, and the responses from the reasoner can be queried by an Xcerpt program. To distinguish ontology queries and responses from other data, and to link queries with their responses, we select a set $\mathcal{O}$ of Xcerpt labels which will be used for this purpose. (Thus we assume that they do not occur elsewhere, particularly in the data queried by programs.) A label from $\mathcal{O}$ will be denoted by #l, or by a sequence of alphanumeric characters beginning with #.

A DIG ask term is a construct term of the form #$l[c_a, c]$, where c_a, c are construct terms. A DIG ask rule is a query rule whose head is a DIG ask term. Such a rule produces data terms of the form #$l[c'_a, c']$. The intention is that c'_a is a DIG ask statement to be send to a reasoner, and c' is additional information to be attached to the reasoner's response. This is the information about the context of the query which the programmer decided to pass.

A DIG response term is a query term of the form #$l[\, q_r, q\,]$, where q_r, q are query terms. The intention is to apply q_r to DIG answers to the questions produced by DIG ask rules with label #l. The query term q is intended to match the additional information attached to the ask statement.

An Extended Xcerpt program is an Xcerpt program containing DIG response rules and DIG ask rules. Syntactically it is just an Xcerpt program. However its semantics is different. The Xcerpt semantics would apply DIG response terms as queries to the the data terms produced by DIG ask rules. Instead, the data terms are used in querying a DIG reasoner, and the DIG response terms query the answers of the reasoner.

A rule $c \leftarrow Q$ directly depends on a rule $c' \leftarrow Q'$, which is not a DIG ask rule, if a query Q'' from Q matches some instance of the construct term c'. Notice that Q'' is not a DIG response term. A DIG response rule $c \leftarrow Q$ directly depends on a DIG ask rule #$l[\ldots] \leftarrow Q'$ if Q contains an DIG response term of the form #$l[\ldots]$. The fact that a rule p directly depends on a rule p' is denoted as $p' \prec p$. A rule p depends on a rule p' in a program P if there exist rules $p_1, \ldots, p_k$ in P such that $k \geq 0$ and $p' \prec p_1 \prec \ldots \prec p_k \prec p$.

A recursive Extended Xcerpt program is a program containing a rule which depends on itself. A program is DIG recursive if it contains a DIG ask rule which depends on itself. Checking whether a program is (DIG) recursive can be done by constructing a dependency graph for the rules in the program. In this paper we define the semantics for programs which are not DIG recursive.

3.1 Operational semantics of Extended Xcerpt

Evaluation of an Extended Xcerpt program P_0 is a sequence of executions of Xcerpt programs and ontology queries. This can be implemented in a rather simple way; an implementation invokes an Xcerpt system and an ontology reasoner with a DIG interface.

We construct a sequence of (non DIG recursive) programs $P_0, \ldots, P_m$, by iteratively employing an ask rule to create DIG queries and replacing the rule by data terms representing the corresponding ontology reasoner answers. The process is repeated until obtaining P_m without ask rules. Evaluation of P_m by Xcerpt produces the result of the initial Extended Xcerpt program P_0.

Program P_{i+1} is obtained from P_i in the following way. Let $p_a = h_a \leftarrow b_a$ be an ask rule from P_i which does not depend on other ask rules.

- *Find results of p_a.* We construct a program P'_i out of P_i by removing all the goal rules and replacing p_a by a goal rule $p'_a = g[\texttt{all}\, h_a] \leftarrow b_a$ where g is a fixed label. (The construct `all` is added to collect all the results of the rule.) The program P'_i is evaluated by the Xcerpt system, producing a result $g[d_1, \ldots, d_n]$. Thus each d_i is a data term of the form #$l[a_i, s_i]$. The subterm a_i should represent a DIG ask statement and s_i is the context information to be returned together with the corresponding answer.

- *Obtain reasoner answers.* Out of the DIG ask statements represented by $a_1, \ldots, a_n$ we build an DIG ask request (which is an XML document additionally containing a header with DIG namespace declarations, and unique identifiers for the elements corresponding to $a_1, \ldots, a_n$). The DIG ask request is sent to the reasoner which replies with a response that (after removing its attributes) can be represented by a data term $responses[r_1, \ldots, r_n]$ where each r_i is a response for a_i.

- *Replace p_a with data terms representing reasoner answers.* For each data term r_i we build a data term $d'_i = $ #$l[r_i, s_i]$. The program P_{i+1} is P_i where the rule p_a is replaced by the set of data terms $\{d'_1, \ldots, d'_n\}$. (Data terms can be represented by rules with empty bodies.)

3.2 Examples

Now we present some examples of Extended Xcerpt programs.

Example 4. Gluten-containing recipes.
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"],
   ingredient[
     name["sugar"],
     amount[ attr{unit["tbsp"]}, 3 ] ],
   ingredient[
     name["orange"],
     amount[ attr{unit["unit"]}, 1 ] ] ],
 recipe[
   name["Recipe2"],
   ingredient[
     name["flour"],
     amount[ attr{unit["dl"]}, 3 ] ],
   ingredient[
     name["salt"],
     amount[ attr{unit["ml"]}, 1 ] ] ],
 recipe[
   name ["Recipe3"],
   ingredient[
     name["spaghetti"],
     amount[ attr{unit["kg"]}, 0.5 ] ],
   ingredient[
     name["tomato"],
     amount[ attr{unit["kg"]}, 0.4 ] ] ]
]
```

Also consider the culinary ingredients ontology from the introduction (Figure 1). We assume that the ontology is loaded into an ontology reasoner with which we can communicate using DIG. We also assume that the names of the ingredients used in the XML document are defined by the ontology. We want to find all the recipes in the XML document which are not gluten-free. This can be achieved using the following Extended Xcerpt program:

```
GOAL
 bad-recipes[ all name[ var R ] ]
FROM
 #gluten[
   true[[ ]],
   name[ var R ]
 ]
END

CONSTRUCT
 #gluten[
   subsumes[
     catom[ attr{ name["gluten-containing"] } ]
     catom[ attr{ name[ var I ] } ] ],
   name[ var R ]
 ]
FROM
 in[ resource[ "file:recipes.xml" ]
    desc recipe[[
      name[ var R ],
      ingredient[[ name[ var I ] ]]
    ]]
 ]
END
```

The program consists of two rules: the first one is a DIG response rule, the second is a DIG ask rule. The ask rule obtains pairs of a recipe name and an ingredient by querying the XML document. It produces ontology queries which ask whether particular ingredients are *gluten-containing*. The additional information attached to the query is the name of a recipe (that uses the ingredient queried about). The recipe name accompanying a query will be attached to the reasoner response to the query; so it is known to which recipe each answer is related.

The responses obtained from the reasoner are queried by the response rule. The responses could be `<true/>`, `<false/>`, or `<error/>`, and the body of the rule matches only `<true/>`. Thus the names of recipes containing at least one *gluten-containing* ingredient are returned by the response rule.

The first phase of evaluation of the program is obtaining the ask statements to be send to the reasoner. The first rule is removed from the program and the second rule changed into a goal rule:

```
GOAL
 g[
  all #gluten[
    subsumes[
     catom[ attr{ name["gluten-containing"]} ]
     catom[ attr{ name[ var I ] } ],
    name[ var R ]
  ]
 ]
FROM
 in[ resource[ "file:recipes.xml" ]
  desc recipe[[
     name[ var R ],
     ingredient[[ name[ var I ] ]]
  ]]
 ]
END
```

The modified program is executed by the Xcerpt system, the result is a data term:

```
g[
#gluten[
 subsumes[
```

```
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["sugar"] } ] ],
 name[ "Recipe1" ] ],
#gluten[
 subsumes[
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["orange"] } ] ],
 name[ "Recipe1" ] ],
#gluten[
 subsumes[
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["flour"] } ] ],
 name[ "Recipe2" ] ],
#gluten[
 subsumes[
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["salt"] } ] ],
 name[ "Recipe2" ] ],
#gluten[
 subsumes[
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["spaghetti"] } ] ],
 name[ "Recipe3" ] ],
#gluten[
 subsumes[
  catom[ attr{ name["gluten-containing"]} ],
  catom[ attr{ name["tomato"] } ] ] ,
 name[ "Recipe3" ] ] ]
```

Out of this data term we build a DIG ask request (which is an XML document), containing six ask statements. The statements are augmented by unique identifiers $1, \ldots, 6$; for instance the first of them is

```
<subsumes id="1">
  <catom name="gluten-containing"/>
  <catom name="sugar"/>
</subsumes>
```

It is remembered which recipe name corresponds to each ask statement.

The DIG ask request is sent to an ontology reasoner. Its XML answer represented by a data term is (the attributes of the element *responses* are removed):

```
responses[
  false [ attr{ id="1" } ],
  false [ attr{ id="2" } ],
  true  [ attr{ id="3" } ],
  false [ attr{ id="4" } ],
  false [ attr{ id="5" } ],
  true  [ attr{ id="6" } ]
]
```

Based on the answer we construct the following set of data terms:

```
#gluten[false[attr{id="1"}],name["Recipe1"]]
#gluten[false[attr{id="2"}],name["Recipe1"]]
#gluten[true [attr{id="3"}],name["Recipe2"]]
#gluten[false[attr{id="4"}],name["Recipe2"]]
#gluten[false[attr{id="5"}],name["Recipe3"]]
#gluten[true [attr{id="6"}],name["Recipe3"]]
```

Now we take the initial Extended Xcerpt program and replace the second rule by the obtained set of data terms:

```
GOAL
  bad-recipes[ all name[ var R ] ]
FROM
  #gluten[
     true[[ ]],
     name[ var R ]
  ]
END

CONSTRUCT
 #gluten[
   false[ attr{id="1"} ],
   name["Recipe1"] ]
END
...
CONSTRUCT
 #gluten[
   true[ attr{id="6"} ],
   name["Recipe3"] ]
END
```

We run this program in Xcerpt obtaining the following final answer:

```
bad-recipes[
  name[ "Recipe2" ],
  name[ "Recipe3" ]
]
```

□

Example 5. Gluten-free recipes (1).

Let us now construct a query producing a list of gluten-free recipes, instead of those containing gluten. This may be less obvious, as we have to take care that none of the ingredients of a recipe contains gluten. One way to find gluten-free recipes is by using the program from Example 4, and extract from `recipes.xml` all the recipes not found by that program to contain gluten. Thus the program from Example 4 is extended with the following rule:

```
GOAL
 good-recipes[ all name[ var R ] ]
FROM
 and[
   in[ resource[ "file:recipes.xml" ]
     desc recipe[[
       name[ var R ],
       ingredient[[ name[ var I ] ]]
     ]] ]
   not bad-recipes[[ name[var R] ]]
 ]
END
```

Also the goal rule of the original program is changed into a construct rule so its results can be queried by other rules:

```
CONSTRUCT
  bad-recipes[ all name[ var R ] ]
FROM
  #gluten[
     true[[ ]],
     name[ var R ]
  ]
END
```

The result of the program is:

```
good-recipes[ name[ "Recipe1"] ]
```

□

Example 6. Gluten-free recipes (2).

The same result as in the previous example can be achieved directly without the intermediate step of finding 'bad-recipes'. For this purpose we can use a program consisting of the ask rule from Example 4 and the following response rule:

```
GOAL
  good-recipes[ all name[ var R ] ]
FROM
  and[
    #gluten[
      var A,
      name[ var R ]
    ],
    not #gluten[
      true[[ ]],
      name[ var R ]
    ]
  ]
END
```

The response rule is used to query reasoner answers which are the same as in Example 4. It selects those recipe names R for which there does not exist an answer *true*.

Xcerpt evaluates first the non negated queries of a rule body. In our case, the query `#gluten[var A, name[var R]]` binds R to the recipe names for which some ontology answers are produced, i.e. "Recipe1", "Recipe2", "Recipe3". (Bindings of variable A to the ontology answers are not used, as A does not occur anywhere else in the rule.) Then, for each value of R, the negated query checks that no ontology answer *true* exists. The check succeeds only for "Recipe1" (as `#gluten[true[[]], name[var R]]` succeeds for R bound to "Recipe2" and to "Recipe3"). Hence the final result of the program is:

```
good-recipes[ name["Recipe1"] ]
```

□

The examples presented so far illustrate usage of filters where Boolean queries are sent to an ontology reasoner. However the presented approach can be used to ask the reasoner arbitrary questions (not only Boolean) which are expressible in DIG.

Example 7.

Consider the ingredients ontology (Figure 1) extended with a class *vitamin*, its three subclasses: *A,B,C* and a property *contained_in*. The extended ontology contains also axioms which indicate in which ingredients a particular vitamin is included. These are the axioms as DIG Tell language statements represented by data terms:

```
impliesc[
  catom[ attr{ name["A"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["tomato"] } ] ] ]
impliesc[
  catom[ attr{ name["B"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["tomato"] } ] ] ]
impliesc[
  catom[ attr{ name["B"] } ] ,
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["orange"] } ] ] ]
impliesc[
  catom[ attr{ name["B"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["flour"] } ] ] ]
impliesc[
  catom[ attr{ name["B"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["spaghetti"] } ] ] ]
impliesc[
  catom[ attr{ name["C"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["orange"] } ] ] ]
impliesc[
  catom[ attr{ name["C"] } ],
  some[
    ratom[ attr{ name["contained_in"] } ],
    catom[ attr{ name["tomato"] } ] ] ]
```

The following Extended Xcerpt program queries the document *recipe.xml* and the ontology to provide a list of vitamins for each recipe in the document. The second rule (ask rule) produces queries to an ontology reasoner which ask about vitamins included in a particular ingredient. The reasoner responses are queried by the first rule (response rule).

```
GOAL
 vit-recipes[ all recipe[var R, all var V] ]
FROM
 #vitamins[
  conceptSet [[
   synonyms[[ catom[attr{ name [var V] }] ]]
  ]],
```

```
  name[var R]
 ]
END

CONSTRUCT
 #vitamins[
   children[
     some[
       ratom[ attr{ name["contained_in"] } ],
       catom[ attr{ name[ var I ] } ]
     ]
   ],
   name[var R]
 ]
FROM
 in[ resource[ "file:recipes.xml" ]
   desc recipe[[
     name[ var R ],
     ingredient[[ name[ var I ] ]]
   ]]
 ]
```

The result of the program is:

```
vit-recipes[
  recipe["Recipe1", "B","C" ],
  recipe["Recipe2", "B"],
  recipe["Recipe3", "A","B","C" ]
]
```

□

3.3 Discussion

We believe that the presented examples illustrate practical usability of the proposed approach. The examples use arbitrary ontology queries, not only Boolean. Notice that there is no restriction on Xcerpt, any its construct can be used. Also there is no restriction on usage of DIG. For instance with a DIG response term `#gluten[[error[[]], name[var R]]]` an Extended Xcerpt program can check, for which data the reasoner returns error.

The ask rules of Extended Xcerpt are used to query an ontology by sending DIG Ask statements to a reasoner. However the rules can be generalized to modify the ontology, by sending DIG Tell statements to the reasoner. In this way ability to modify ontologies could be added to Extended Xcerpt.

The operational semantics that we presented (Section 3.1) requires that Extended Xcerpt programs are non DIG recursive. The restriction was chosen to simplify the semantics. We expect that a generalization to arbitrary programs is not difficult. It is not clear whether DIG recursive programs are needed in practice.

The semantics of Extended Xcerpt imposes certain implicit type requirements on programs. The data terms produced by a DIG ask rule should be of the form $\#l(a, s)$, where a is a DIG ask statement (represented as a data term). A DIG response term should match data terms of the form $\#l(r, s)$, where r is a DIG response statement (as a data term). Moreover, r is a response for some ask statement a such that some some rule in the program is able to produce $\#l(a, s)$. It is better to check such conditions statically, instead of facing run-time errors. For this purpose the descriptive type system for Xcerpt [3, 12] can be used.

The presented extension of Xcerpt requires the programmer to use DIG syntax for ontology queries. Also, one has to write ask rules, which create DIG queries, and response rules, which deal with DIG responses. Such approach may be considered as rather low level. One may prefer augmenting Xcerpt rules with ontology queries, and using rules of the form $c \leftarrow O, Q$, where c is a construct term, O an ontology query and Q an Xcerpt query. We expect that a convenient way of implementing such generalization of Xcerpt is compiling it to Extended Xcerpt.

We expect that the approach of this paper can be applied to composing some other XML query languages with ontology querying.

4 Conclusions

The paper addresses the problem of how to use ontological information in the context of querying XML data. The proposed solution extends the XML query language Xcerpt by adding the possibility of querying ontologies. Programs in Extended Xcerpt communicate with an ontology reasoner using DIG interface. No restrictions are imposed on the Xcerpt language and on the DIG ask statements used. In particular, ontologies can be queried with arbitrary, not only Boolean, queries. Data obtained from ontology querying can be used in XML querying, and vice versa. An implementation of Extended Xcerpt can employ an existing Xcerpt implementation and an existing ontology reasoner; they are treated as "black boxes" (no modifications to the Xcerpt system or the reasoner are needed).

References

[1] U. Aßman, J. Henriksson, and J. Małuszyński. Combining Safe Rules and Ontologies by Interfacing of Reasoners. In *International Workshop, PPSWR 2006, Budva, Montenegro, June 2006, Proceedings*, pages 31–43, 2006.

[2] S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. In *Proceedings of DL2003 Workshop*, Rome, 2003.

[3] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive typing rules for Xcerpt. In *International Workshop, PSWR 2005, Dagstuhl Castle, Germany, September 2005, Proceedings*, number 3703 in LNCS, pages 85–100. Springer Verlag, 2005.

[4] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Al-log: integrating datalog and description logics. *J. of Intelligent and Cooperative Information Systems*, 10:227–252, 1998.

[5] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *International Conference of Knowledge Representation and Reasoning*, 2004.

[6] A. Y. Levy and M.-C. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In *European Conference on Artificial Intelligence*, pages 323–327, 1996.

[7] R. Rosati. Semantic and computational advantages of the safe integration of ontologies and rules. In *International Workshop, PPSWR 2005, Dagstuhl Castle, Germany, September 2005, Proceedings*, volume 3703, pages 50–64, 2005.

[8] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, Germany, 2004.

[9] 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.

[10] E. Svensson and A. Wilk. XML Querying Using Ontological Information. In *International Workshop, PPSWR 2006, Budva, Montenegro, June 2006, Proceedings*, pages 187–199, 2006.

[11] W3 Consortium. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`.

[12] A. Wilk. Descriptive Types for XML Query Language Xcerpt. Technical report, Linköping universitet, Sweden, 2006.