Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 235 (2009) 19–33
www.elsevier.com/locate/entcs

# An Ontology-based System for Semantic Filtering of XML Data*

M. Baggi[1]

*Dip. di Scienze Matematiche e Informatiche*
*Pian dei Mantellini 44, 53100 Siena, Italy.*

Abstract

In this paper, we describe a system, written in Haskell, for the ontology-based approximate filtering of XML data. Basically, it allows one to extract relevant data as well as to exclude useless and misleading contents from an XML document. The system provides a declarative language one can use to define XML patterns and ontology queries to express syntactic as well as semantic filtering criteria. The system can be used through a Web application which is endowed with a user-friendly graphical interface. Finally, we provide some meaningful examples which show the usefulness of the implemented filtering methodology.

*Keywords:* XML, Haskell, ontology-based approximate filter

## 1 Introduction

The adoption of the XML[20] language, designed by the World Wide Web Consortium (W3C), for representing semistructured data on the Web, has lead to a rapid growth in the amount of XML data available over the internet. Taking advantage of the huge amount of implicit and distributed information on the Web is a significant challenge.

To face up such an information overload, in the last years a lot of efforts have been invested to develop query and filtering languages as means to consult XML documents. The W3C has defined some standard languages to query XML contents, such as XQuery[22] and XPath[21]. However, a large variety of proposals have been developed independently, e.g. [10,17,14]. All these solutions are characterized by an *exact* matching mechanism of a given pattern (or path expression) against an XML document, which then delivers to the user all the recognized pattern instances.

* This work has been partially supported by the italian MUR under grant n. RBIN04M8S8, FIRB project, internationalization 2004.
[1] Email: baggi@unisi.it


doi:10.1016/j.entcs.2009.03.003

Although these approaches are very advantageous in many applications, they might not be much suitable when dealing with data filtering in a pure information retrieval context, since *(i)* they require the user to be aware of the complete XML document structure, *(ii)* results that are not exactly matched are discarded and *(iii)* they only rely on the structure of the document (that is, the *syntax* of the data), hence *semantic* filtering is not allowed.

XML data can be equipped with a semantics formalized by a given ontology which enriches data with meanings and properties. Therefore, using ontologies to explicitly represent domain-specific knowledge allows one reasoning about the XML content under analysis. By exploiting such ontological information it is possible to improve the filtering mechanism.

Some programming languages supporting XML processing have also been developed, such as XCentric [13] which is a logic language, extending Prolog with a richer form of unification and regular types, designed specifically for XML processing in logic programming. XCentric is also employed as part of VeriFLog [12] which is a tool for verification of web sites content and data inference. CDuce and XDuce ([8], [16]) are typed functional programming languages, based on pattern matching, designed to support XML applications. Such languages can be used to consult and query XML documents but provide basically an exact matching behavior and the semantic information is not considered.

In [6] is presented a rewriting-like methodology to filter information from XML documents where the positive and negative filtering concepts are introduced. The work proposed in [4] extends the ideas in [6] introducing approximate pattern-matching and presents a declarative pattern-based language for XML filtering, endowed with an approximate pattern-matching engine. A further improvement is proposed in [5] where we have integrated the approximate pattern-matching engine with ontology reasoning capabilities in order to enable semantic data filtering.

In this paper, we describe XPhil, an implementation of such a language. Within the XPhil methodology, patterns are searched in an XML document in an approximate way (that is, modulo renaming, deletion and insertion of XML items) using additional information which can be retrieved by querying (possibly remote) ontologies, formalized by Description logic [3].

The already mentioned XQuery language, which uses XPath to describe path expressions, provide a minimal degree of flexibility. For example, element insertions in path expressions are only allowed by the explicit use of the \\ operator, while no element deletion or renaming is allowed. Furthermore, no semantic based filtering facilities are provided.

In our framework, the connection between the filtering engine and the ontology reasoner is realized by means of (an extended version of) the DIG interface [7], which is a standard API for description logic systems supported by a number of ontology reasoners (e.g. Pellet, RacerPro).

The problem of combining XML languages with ontology reasoning seems to be important for the Semantic Web. Recently, many efforts have been made to improve the existing syntax-guided query languages to deal with semantic informa-

tion and many new approaches have been proposed. One of them, quite similar to ours, is DIGXcerpt[15], which presents a language that extends the XML query language Xcerpt[11] enhancing structural querying of XML data with ontology reasoning. Similarly to our framework, they employ DIG to interface the XML query engine with the ontology reasoner. Although DIGXcerpt allows a powerful search and manipulation of XML data, it only performs exact query matching, while our methodology is able to perform a more flexible matching which ranks the query results w.r.t. computed similarity degree.

The Resource Description Framework (RDF)[24] has been designed to complement XML data with semantic annotations. A large number of prototype systems able to read and reason about such annotations have been developed, for example [2], which describes a logical framework in which XQuery programs are enriched with RDF metadata. This approach enables a limited form of data inference from RDF documents, which is not as powerful as the reasoning capabilities of more complex description logic formalisms like the one supported by the DIG interface.

**Plan of the paper.** The rest of the paper is organized as follows. Section 2 introduces the DIG interface languages, by means of which we model ontologies and ontology queries; then, we provide an extension of the standard DIG query language which is employed to define ontology query templates. Section 3 presents our filtering language which combines an approximate pattern-matching engine with an ontology reasoning mechanism. In Section 4, we describe the XPhil system architecture and the way it works in order to answer user queries. In Section 5, we briefly describe the Web interface by means of which the system can be tested. Finally, Section 6 concludes.

## 2 Modeling and Querying Ontologies

As mentioned above, the DIG interface is an API for description logic (in short, DL) systems which is capable of expressing class and property expressions common to most DLs. In particular, it can model the well-known description logic formalized within the OWL-DL[23] framework, which is supported by several ontology reasoners.

The DIG interface is equipped with four XML languages which are employed to formalize and query ontologies modeling a given application domain. These languages are (i) the *tell* language, (ii) the *concept* language, (iii) the *ask* language, and (iv) the *response* language. [2]

The DIG *concept* and *tell* languages basically contain constructs for describing and then loading an ontology into a reasoner. Roughly speaking, they allow us to formalize the structure of an ontology by defining concepts (classes), roles (relations), individuals (instances of classes), *etc.*

For instance, Figures 1 illustrates the (graph) structure of an ontology over the domain of an academic structure. In the following we refer to this ontology with

[2] The complete DIG formalization is available at [5].

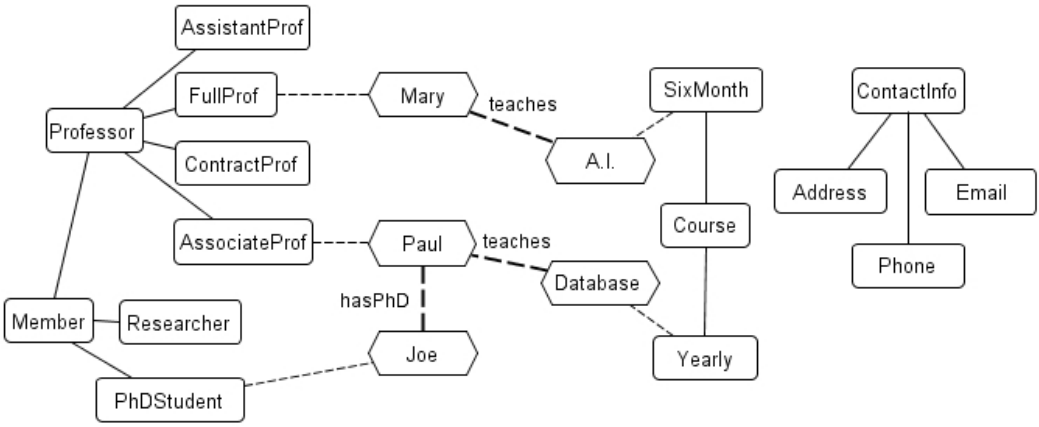


Figure 1. An ontology about an academic structure

the name `univ`. Such ontology can be easily defined by using the DIG formalism, more precisely by means of the DIG *tell* language which is devoted to ontology description. The `univ` ontology models people and courses as individuals. Each person has a position inside the structure, like associate professor, researcher, PhD student and so on. Positions are represented by means of hierarchically organized concepts.

In Figures 1, we have used rectangles to denote concepts, while hexagons denote individuals. Straight, solid lines connecting concepts represent subconcept relations (e.g. "a full professor is a professor"). Dashed lines connecting individuals to concepts define membership relations of individuals w.r.t. concepts (e.g. "the individual Mary belongs to the full professor concept"). Finally, bold, dashed lines connecting pairs of individuals formalize roles, that are, binary relations between individuals (e.g. "the associate professor Paul has a PhD student Joe", or "the full professor Mary teaches a six month course named A.I.").

The DIG interface, whose complete formalization is available at [5], includes an *ask* language whose constructs are used to query ontologies loaded into ontology reasoners. *Ask* statements allow us to infer information regarding concepts, roles and individuals of a given ontology. Moreover, they can model boolean as well as non-boolean ontology queries. More precisely, a boolean (respectively, non-boolean) ontology query is an *ask* statement that returns a boolean (respectively, non-boolean) value.

Finally, the DIG *response* language formalizes the possible response statements generated after the execution of an *ask* statement (e.g. boolean values, sets of ontology elements, error messages,...)

### *2.1 DIG Ask Language Extension*

DIG *ask* statements are basically ground formulae –that is, formulae not containing variables– of a given description logic. In order to make them more flexible and suitable for our purposes, we adopted a generalized version of *ask* statements, by

defining "templates" which
*(i)* can be easily reused in several filtering rules;
*(ii)* can be instantiated with values computed at run-time.
Therefore, we extend the DIG *ask* language by

- introducing variables into *ask* statements. Variables are employed as placeholders for concepts, roles, and individuals. We denote a variable, whose name is `varName`, by the syntax `var:varName`. *Ask* statements containing variables are called *non-ground ask* statements.
- letting *ask* statements denote non-boolean queries to reference XML tags of a given XML document via the `tag:self` notation. As we will show in Section 3, a non-boolean ontology query `Q` is typically bound to an XML tag `t`. By using the `tag:self` construct, we can automatically reference `t` inside `Q` without citing it explicitly.

For the sake of readability, in the following examples and throughout the whole paper, we omit namespace declarations inside DIG statements.

**Example 2.1** Consider the ontology of Figure 1. The following non-ground *ask* statement models a boolean query which checks whether (the value assigned to) the variable `X` is a six month course. Roughly speaking, the query verifies whether the value bounded to `X` is an instance of the concept `SixMonth`.

```
<asks xmlns=... >
 <instance id="HalfYearlyCourse">
  <individual name="var:X"/>
  <catom name="SixMonth"/>
 </instance>
</asks>
```

**Example 2.2** Referring to ontology of Figure 1, the following non-ground *ask* statement models a boolean query which checks whether (the value assigned to) the variable `X` is an instance of the concept `Professor`.

```
<asks xmlns=... >
 <instance id="Prof">
  <individual name="var:X"/>
  <catom name="Professor"/>
 </instance>
</asks>
```

**Example 2.3** Consider again the ontology of Figure 1. The following non-ground *ask* statement models a boolean query which checks whether (the value assigned to) the variable `Y` is an associate professor having more than three PhD students. This is formalized defining a complex concept which is an intersection of two concepts. The former is `AssociateProf` while the latter is obtained by projecting the role `hasPhD` on its first component with the constraint that each professor has to be related to at least three instances of the concept `PhDStudent`.

```
<asks xmlns=... >
 <instance id="BusyProf">
  <individual name="var:Y"/>
  <and>
   <catom name="AssociateProf"/>
   <atleast num="3">
    <ratom name="hasPhD"/>
    <catom name="PhDStudent"/>
   </atleast>
  </and>
 </instance>
</asks>
```

**Example 2.4** Assume that a given XML tag `t` is associated with the following *ask* statement modeling a non-boolean query `Q`.

```
<asks xmlns=... >
 <children id="Syn">
  <catom name="tag:self"/>
 </children>
</asks>
```

The query `Q` retrieves all the concepts which are children of the concept `t`.

## 3 Combining Ontology Reasoning with XML Filtering

The filtering language we propose is a declarative, pattern-based language in which filtering rules can be specified as (possibly conditional) XML patterns. A filtering rule matches an XML document if the pattern is somehow "embedded" into the XML document and fulfills the specified conditions. We adopt an approximate pattern-matching mechanism, which we have formalized in [4]. Roughly speaking, it is based on a cost-based pattern transformation algorithm which searches for patterns in an approximate way and ranks the results w.r.t. their cost. Pattern transformations, which may consists in deleting, inserting or renaming XML items, minimally modify the original pattern and adapt it to the XML document with the aim of finding the best matches. Each pattern transformation has a cost, represented by a natural number. The overall matching cost, which is used to rank the filtered information, is tightly bound to the pattern transformations needed to find an approximate match. Therefore, an exact match would be associated with a null cost. A detailed description of the algorithm can be found in [4].

In the following, we briefly describe the filtering language integrating filtering rules with extended DIG *ask* statements used to infer information from ontologies. The benefits of such additional information are: (i) automatizing the search of the XML tag synonyms employed by the approximate matching engine when renaming pattern transformations are applied; (ii) using boolean ontology queries as filtering conditions to refine pattern detection.

### 3.1 The Extended Filtering Language

A filtering rule can be formalized by means of the following XML syntax [3]:

```
<rule>
 {<count cost=" value"/>}
 <filter> filterop </filter>
 <pattern> XML-pat </pattern>
 <document> XML-doc </document>
 {<conditions> cond-list </conditions>}
 {<mode> mode </mode>}
</rule>
```

which informally says that

1. a pattern *XML-pat* is searched in a document *XML-doc*;
2. only detected instances of *XML-pat* which satisfy the given list of conditions *cond-list* are either extracted (*positive filtering*) or removed (*negative filtering*) from *XML-doc* according to the value of the filtering mode *mode*. A filtering mode is a label in the set {P,N}.
3. count is an optional operator which allows us to count the number of detected pattern instances with similarity cost less than *value* in the given XML document.

*XML-doc* can be specified by an URL referring to an XML document, by some XML code, or even by a nested filtering rule, since the execution of a filtering rule generates an XML document, which can feed another filtering rule.

Moreover, we provide four filtering operators *filterop* which can model both exact and approximative filtering w.r.t. a universal as well as existential semantics (i.e. filter{One,All}Exact, filter{One,All}Best). Finally, note that, when no conditions are specified, the <conditions> tag of a filtering rule can be omitted.

**XML Patterns.** Filtering rule patterns are XML data used to describe the structure of the information we want to detect inside a given XML document. A tag selector (e.g. <professor>) is an XML pattern which can contain

- another tag selector (e.g. <professor><name></name></professor>);
- a boolean tag connector among <and>, <or> and <xor>, which are used to express conjunctions and disjunctions of XML patterns;
- a variable, that plays the role of placeholder for an unknown piece of XML code (e.g. <name>X</name>);
- a text selector, that is a strings of plain text surrounded by single quotes (e.g. <name>'John'</name>).

Note that boolean tag connectors may be used also at the root level of an XML pattern.

[3] The extended filtering language is defined by an XML Schema which is available at [1].

Moreover, tag selectors can contain three different attributes. The first two attributes `ont` and `query` enable tag flexible matching (i.e. matching modulo tag renaming). The `ont` attribute specifies an ontology file name, while the `query` attribute specifies the file name of an extended DIG *ask* statement modeling a non-boolean ontology query. More precisely, given a tag selector `t`, by the syntax `<t ont="ontName" query="queryName">` we retrieve all the synonyms of the tag selector `t` by querying the ontology `ontName` via the query `queryName`. Such synonyms are then used by the pattern-matching algorithm to find approximate results. As we have seen in Section 2, the tag selector `t` can be referenced inside the query `queryName` using the `tag:self` notation. During the query execution `tag:self` occurrences are replaced by `t`.

**Example 3.1** Consider the ontology of Figure 1, and the extended *ask* statement of Example 2.4. Then, the synonyms retrieved for the tag selector
`<professor ont="univ" query="Syn">`
are `{AssociateProf,FullProf,ContractProf,AssistantProf}`.

The third attribute of tag selectors is *child* whose value is an element of the set $\mathbb{N} \cup \{\text{last}\}$. Given an XML document containing a tag `s` that has $n$ children labeled by tag `t`, the notation `<t child="i">` selects the `i`-th child labeled with `t`. The keyword `last` is used to select the last element of the sequence.

**Example 3.2** Consider the XML document

```
<professors>
 <professor>Mary</professor>
 <professor>Paul</professor>
</professors>
```

Then, `<professors><professor child="2">X</professor></professors>` is an XML pattern selecting the piece of XML
`<professors><professor>Paul</professor></professors>` (as a side effect the matching mechanism binds variable `X` to `Paul`).

**Conditions.** Conditions are employed to further refine the search of a given pattern inside an XML document. When some instances of the XML pattern have been detected inside the XML document and variables have been bounded to some values, the associated instance of the condition list is evaluated and the pattern instance is then delivered to the user if and only if the condition list instance evaluates to true.

Our language allows one to specify three classes of conditions:
*Membership tests* of the form `X in rexp` which allow one to establish whether the piece of XML associated to a variable is contained in the language denoted by a given regular expression (`rexp`)[4]. If the variable is bound to a complex XML subtree (not just a textual node), we build up a string `s` concatenating the labels of all the textual nodes in the subtree, traversing it from left to right, and we subsequently check whether `s` belongs to the language denoted by the considered

[4] Regular languages are represented by means of the usual Unix-like regular expressions syntax [18].

regular expression `rexp`.
*Functional constraints* which allow one to perform some computations over the extracted XML data and then check the results.

These two kinds of conditions are formalized using a simplified version of RuleML [9] (the syntax details can be found in the XML Schema available at [1]).

**Example 3.3** Assume that an XML document modeling university courses is given. For each professor we know both the name and the surname. For each course, the professor giving the course is identified by the concatenation of his/her surname and the initial of his/her first name. Using three variables `X`, `Y`, and `Z`, we can model such property by means of the following functional constraint `X++first(Y)=Z`. Following the RuleML syntax, such condition will be expressed by means of the following XML code.

```
<Equal>
 <lhs> <Expr>
        <Fun>+</Fun>
        <Var>X</Var>
        <Expr> <Fun>first</Fun> <Var>Y</Var> </Expr>
       </Expr>
 </lhs>
 <rhs> <Var>Z</Var> </rhs>
</Equal>
```

*Semantic constraints* which allow one to check semantic properties of XML documents. Semantic constraints are specified through boolean ontology queries, which are then executed against a given ontology.

Semantic constraints are formalized by means of the following syntax `<ontCond ont="ontName" query="queryName">` where `ontName` is an ontology file name, and `queryName` represents the file name of an extended DIG *ask* statement modeling a boolean ontology query. Note that extended DIG *ask* statements may contain variables, hence, before sending the statements to the reasoner, such variables are instantiated with values of the detected pattern instances.

**Example 3.4** Consider the ontology of Figure 1 and the extended *ask* statement of Example 2.3 which models the boolean query `BusyProf`. Therefore, the following semantic constraint `<ontCond ont="univ" query="BusyProf">` evaluates to true if and only if the value bound to variable `Y` is an associate professor having more than three PhD students.

**Example 3.5** Consider an XML repository containing academic information along with the ontology `univ`. Suppose we want to search for *busy* associate professors, that are, associate professors who are titular of at least one six month course and have more than three PhD students. Then, a possible rule searching for this property might be

```
<rule>
 <filter>filterAllExact</filter>
```

```
 <pattern>
  <name>X</name>
 </pattern>
 <document>
  <nestRule>
   <rule>
     <filter>filterAllBest</filter>
     <pattern>
       <course>
        <and>
         <id>X</id>
         <titular>
          <name>Y</name>
         </titular>
        </and>
       </course>
     </pattern>
     <document>
       <docFile>'courses.xml'</docFile>
     </document>
     <conditionClause>
       <ontCond ont="univ.xml" query="BusyProf.xml"/>
       <ontCond ont="univ.xml" query="HalfYearlyCourse.xml"/>
     </conditionClause>
   </rule>
  </nestRule>
 </document>
</rule>
```

**Example 3.6** Consider the same XML repository and the `univ` ontology. Suppose we have an XML document containing personal information about professors and we want to query it in order to obtain all contacts information for young professors (that is, professors whose age is less than 40). A query formalizing this property might be as follows.

```
<rule>
 <filter>filterAllBest</filter>
 <pattern>
   <member>
    <and>
     <name>X</name>
     <age>Y</age>
     <contactInfo ont="univ.xml" query="Syn.xml">Z</contactInfo>
    </and>
   </member>
 </pattern>
```

```
 <document>
  <docFile>'members.xml'</docFile>
 </document>
 <conditionClause>
  <ontCond ont="univ.xml" query="Prof.xml"/>
  <opCond>
   <Predicate>
    <Rel> lt </Rel>
    <lhs> <Var>Z</Var> </lhs>
    <rhs> <Data>40</Data> </rhs>
   </Predicate>
  </opCond>
 </conditionClause>
</rule>
```

## 4 System architecture

The implemented system is divided into three parts: the front-end, the core and the ontology reasoner. The core can be in turn divided in a controller, an approximate matching engine, an ontology interface and a condition evaluator. This architecture is represented in Figure 2. Users may use the front-end to load XML documents, ontologies and ontology queries, specify and execute the querying rules and view the results. The core is responsible for the execution of rules by means of the controller that manages the interaction of the matching engine, the condition evaluator and the ontology interface. The matching engine is aimed to find the best approximate matches of the rule pattern against the given XML document, the condition evaluator deals with the evaluation of rule conditions and the ontology interface allows one to load and release ontologies in the ontology reasoner and send ontology queries to the ontology reasoner. In our implementation we have used Pellet[19] as an efficient, open-source ontology reasoner for the OWL-DL[23] framework.

When a rule is executed, the controller locates into the rule the required ontologies and asks the ontology interface to load such ontologies in the reasoner. Then, the approximate matching engine is activated to find the best matches of the pattern against the specified XML document. During the matching phase, whenever a tag selector associated with an ontology query is processed, the ontology interface is called to query the appropriate ontology and return the results. Whenever one or more embeddings of the rule pattern are found in the XML document, the rule conditions (if any) have to be verified. The only delivered results will be those embeddings whose condition instances evaluate to true. Membership tests and functional constraints are managed by the condition evaluator, while to assess semantic conditions, the ontology interface need to be called to transfer the query to the ontology reasoner. Finally the controller asks the ontology interface to release all ontologies, packs the results and sends them to the front-end.

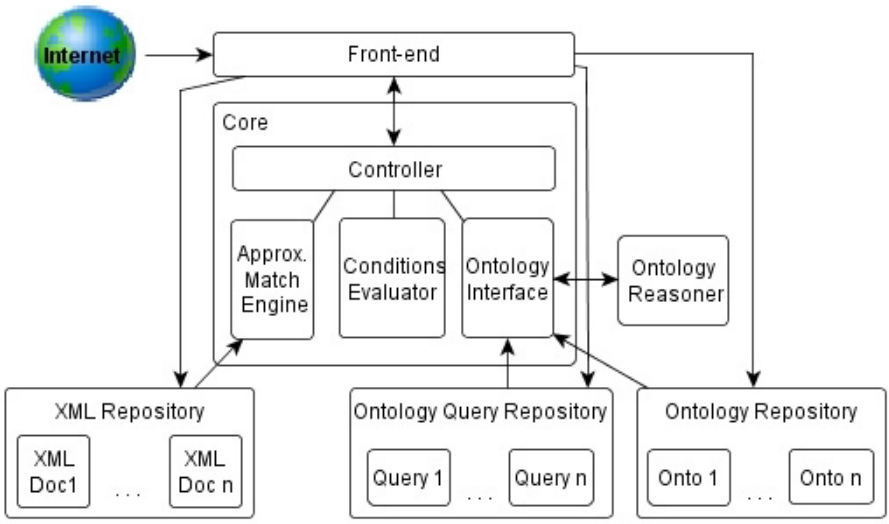


Figure 2. System Architecture

## 5 The XPhil Filtering System

The filtering language has been implemented in Haskell (GHC 6.6), the sources and executable files are publicly available together with a set of examples at

`http://users.dimi.uniud.it/~michele.baggi/xphil.html`

while an online demo is available at

`http://users.dimi.uniud.it/~michele.baggi/XPhilWS/index.php.`

The implementation contains the context-free grammar of our language and some parsers for the filtering language and for XML documents. There are some modules for the approximate pattern-matching algorithm and one module to manage the variable associations, one module to interface ontology reasoners and another one to evaluate syntactic and semantic rule conditions.

The implementation has been written in the Haskell functional language whose lazy evaluation engine is able to make only use of the information strictly needed to evaluate expressions. Hence, only the portion of the data structures which are strictly necessary to evaluate a filtering rule are generated with a consequent gain in the overall system performance. For further details on the system performance, please refer to [5].

Figure 3 provides a snapshot of the graphical user interface of the XPhil online system. The interface allows one to load and execute some example rules or build and execute user-defined rules. The interface is divided in six panels whose functions are briefly explained in the following:

- The *Example Rule* panel contains some links to load some rule examples.
- The *XPhil Rule* panel, showing at the beginning the message *No Rule Loaded*, contains a text area where users can edit a rule, otherwise it is possible to load an XML file containing a rule using the *Browse* button. The *Load Rule* button

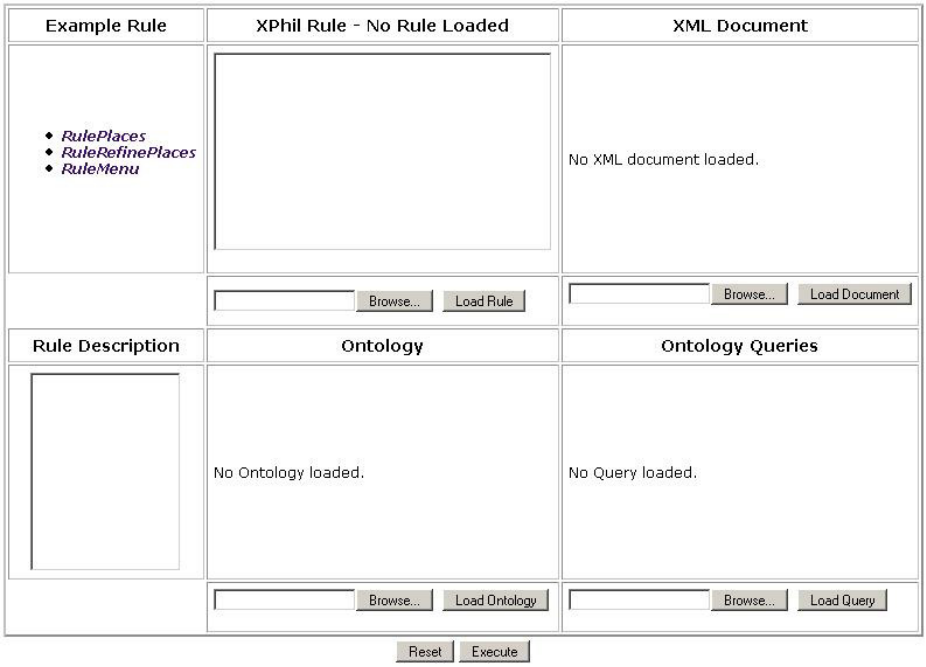


Figure 3. Screenshot of the XPhil system online.

loads the edited or browsed rule into the system and the *No Rule Loaded* message will change into *Rule Loaded*.

- The *XML Document* panel allows one to search an XML document browsing the file system using the *Browse* button and then loading it by means of the *Load Document* button.
- The *Rule Description* panel is usually empty. When an example rule is loaded, this panel may contain a brief rule explanation in natural language.
- The *Ontology* panel allows one to load one or more XML files containing an ontology description specified by means of DIG *tell* statements. The *Browse* button allows one to browse the file system and the *Load Ontology* button loads the specified ontology.
- The *Ontology Queries* panel allows one to load one or more XML files, containing extended DIG *ask* statements modeling boolean ontology queries. The *Browse* button allows one to browse the file system and the *Load Query* button will load the specified query.
- Finally, the *Reset* button is used to clean up all the panels while the *Execute* button executes the specified rule and the result will be displayed in a new window.

## 6 Conclusions

In this paper we presented the XPhil system, which implements an approximate filtering language combining pattern matching with ontology reasoning in order to

enable semantic data filtering. More precisely, it allows us to search XML patterns into XML documents w.r.t. semantic criteria.

The filtering process is performed by an approximate pattern-matching engine which queries an ontology reasoner to infer semantic information regarding the XML data. Semantic information are employed (i) to automatize the search of XML tag synonyms used by the matching mechanism when performing renaming operations; (ii) to model semantic properties of the data to be extracted.

Moreover, we described the architecture of the implemented system and an online user friendly graphical interface through which the system can be used. Since the developed interface still requires users to write filtering rules using a rather verbose XML syntax, we are planning to build an intuitive visual interface for editing filtering rules which avoids to manage tedious XML details and error-prone textual descriptions.

## References

[1] The eXtended Filtering Language XPhil, 2008. Available at http://users.dimi.uniud.it/~michele.baggi/xphil.html.

[2] J.M. Almendros. A RDF Query Language Based on Logic Programming. In *Proc. of the 3rd Int'l Workshop on Automated Specification and Verification of Web Systems (WWV'07)*. ENTCS, Elsevier, 2008. to appear.

[3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Scheider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[4] M. Baggi and D. Ballis. PHIL: A Lazy Implementation of a Language for Approximate Filtering of XML Documents. In *Proc. of 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07)*, 2007.

[5] M. Baggi, D. Ballis, and M. Falaschi. XML Semantic Filtering via Ontology Reasoning. In *Proc. of 3rd International Conference on Internet and Web Applications and Services (ICIW'08)*. IEEE Computer Society Press, 2008. to appear.

[6] D. Ballis and D. Romero. Filtering of XML Documents. *2nd Int'l. Workshop on Automated Specification and Verification of Web Systems (WWV'06)*, 0:19–28, 2006.

[7] S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. Technical report, University of Manchester, 2003.

[8] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proc. of 8th ACM SIGPLAN International Conference on Functional Programming, (ICFP'03)*, pages 51–63, 2003.

[9] Harold Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. of 14th International Conference on Applications of Prolog, (INAP'01)*, volume 2543 of *Lecture Notes in Computer Science*, pages 5–22, 2001.

[10] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of the Int'l Conference on Logic Programming (ICLP'02)*, volume 2401 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[11] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[12] J. Coelho and M. Florido. VeriFLog: A Constraint Logic Programming Approach to Verification of Website Content. In *APWeb International Workshops*, pages 148–156, 2006.

[13] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *Proc. of 9th ACM International Workshop on Web Information and Data Management, (WIDM'07)*, pages 1–8, 2007.

[14] A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and Abstract Semantics of a Graphical Query Language. *Theoretical Computer Science*, 275:521–560, 2002.

[15] W. Drabent and A. Wilk. Extending XML Query Language Xcerpt by Ontology Queries. In *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, pages 447–451. IEEE Computer Society Press, 2007.

[16] H. Hosoya and B.C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[17] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 2004.

[18] The Open Group. Unix Regular Expressions. Available at: http://www.opengroup.org/onlinepubs/7908799/xbd/re.html.

[19] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[20] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: http://www.w3.org.

[21] World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: http://www.w3.org.

[22] World Wide Web Consortium (W3C). XQuery: A Query Language for XML, 2001. Available at: http://www.w3.org.

[23] World Wide Web Consortium (W3C). OWL Web Ontology Language Guide, 2004. Available at: http://www.w3.org/.

[24] World Wide Web Consortium (W3C). RDF Vocabulary Description Language 1.0: RDF Schema, 2004. Available at: http://www.w3.org/.