Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Applied Logic 3 (2005) 253–270
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Rule-based schema matching for ontology-based mediators

Gunter Saake [a,*], Kai-Uwe Sattler [b], Stefan Conrad [c,1]

[a] *University of Magdeburg, Computer Science, Universitätsplatz 2, D-39106 Magdeburg, Germany*
[b] *Technical University of Ilmenau, Computer Science and Automation, P.O. Box 100 565, D-98684 Ilmenau, Germany*
[c] *University of Düsseldorf, Computer Science, Universitätsstr. 1, D-40225 Düsseldorf, Germany*

Available online 11 September 2004

Abstract

Mediating heterogeneous data sources heavily relies on explicit domain knowledge expressed, for example, as ontologies and mapping rules. We discuss the use of logic representations for mapping schema elements onto concepts expressed in a simplified ontology for cultural assets. Starting with a logic representation of the ontology, criteria for a rule-based schema matching are exemplified. Special requirements are the handling of uncertain information and the processing of hierarchical XML structures representing instances.

Keywords: Schema matching; Data integration; Logic representation

1. Introduction

Integrating data from heterogeneous sources on the Web is an important topic of interest within the database community. Current approaches try to overcome limitations of the first structural oriented mediator generation by explicit modeling and usage of domain knowl-

* Corresponding author.
E-mail addresses: saake@iti.cs.uni-magdeburg.de (G. Saake), kus@tu-ilmenau.de (K.-U. Sattler), conrad@cs.uni-duesseldorf.de (S. Conrad).
[1] This work was partly supported by DFG (Deutsche Forschungsgemeinschaft): CO 207/13-1/3.

1570-8683/$ – see front matter
doi:10.1016/j.jal.2004.07.021

edge in form of semantic meta data, i.e., a vocabulary, a taxonomy, a concept hierarchy, or even an ontology. However, a special requirement from data integration is still to define a mapping from the ontology layer to the source data, i.e., to specify how a data source supports a certain concept from the ontology both in a structural as well as in a semantic way. This correspondence information is necessary for query rewriting and decomposition and has to be provided as part of the registration of a source.

There are several possible ways for specifying schema correspondences. In the global-as-view approach (GAV) the global mediator schema is defined as view on the local schemas. In contrast, the local-as-view approach (LAV) starts with the global schema and defines the local schemas as views on it. Here, local sources are modeled always as a subset of the global schema as well as the class extensions. GAV results in simpler query processing because the query rewriting step requires only a view resolution—as long as no global integrity constraints have to be taken into account. Otherwise, query processing becomes more complex as shown in [1].

On the other side, the LAV principle simplifies adding or removing sources because only correspondences between the global schema and the particular local schema have to be considered. A detailed discussion of issues on LAV vs. GAV is given for example in [1]. In [2] the authors propose a GLAV approach—a combination of both approaches allowing a more flexible mapping definition.

In any case, specifying the mapping by hand is an expensive and error-prone process, especially for complex schemas and/or ontologies. Schema matching approaches [3] try to reduce the effort by comparing schemas of different sources and identify matchings based on structural correspondences and—to a certain degree—by exploiting information about the actual data.

In the paper, we argue that these approaches can be improved by using declarative rules which are used during matching, even if correspondences are “hidden” due to different names of classes and attributes and can deal with sub-class hierarchies which are often used for modeling ontologies. This leads to extensible matchers allowing to add user-specified rules which could be even derived from already existing correspondences. In this way, domain-specific rules for matching certain elements by exploiting background knowledge or for combining different matchers in a specific manner can be easily added without modifying or bloating the matching tool.

We present the logic-based foundations of this approach, discuss several schema matching rules and their composition for defining matches between the ontology level representing the global mediator schema as well as the source schemas. Finally, we discuss the application of this approach for specifying mappings for our ontology-based mediator system YACOB. The approach presented here as well as the accompanying tool support are currently still under development. Therefore, we focus in this paper on the presentation of the basic ideas leaving details, such as considering instance level information, for future work.

2. Related work

Schema matching is an important subtask of data integration. The core of schema matching is the operator *Match* which takes two schemas as input and produces a mapping

between the elements of these schemas based on semantic correspondences. Implementing this operator requires an internal representation to which imported schemas are translated and which allows a generic solution. This can be further supported by using dictionaries, thesauri and other kind of domain knowledge useful for identifying correspondences.

In most cases, schema matching cannot be done fully automatically—often some kind of user intervention or decision is required in order to accept, modify or reject matchings found by the system. Nevertheless, several approaches and tools were developed for supporting schema matching in a semi-automatic way which combine techniques from schema translation, graph transformation, machine learning and knowledge representation. A good survey of these approaches is given in [3]. Here, we briefly summarize this work and discuss it with regard to a rule-based approach.

In [3] the authors classify schema matching approaches into three classes:

- *individual matchers* compute a mapping using only a single match criterion,
- *hybrid matchers* support multiple criteria by using a fixed combination of individual matching techniques [4],
- *composite matchers* combine the results of individual matchers depending on schema characteristics, application domain or even results of previous steps, e.g., by applying techniques from machine learning [5].

Individual matchers as building blocks for hybrid and composite matchers can be further classified into:

- *Schema vs. instance level*: Schema-level matchers consider only schema information such as structures (data types, classes, attributes) as well as properties of schema elements like name, type etc. In contrast, instance-level matchers consider data contents, too. This allows a more detailed characterization of data, especially in cases with incomplete or unknown schema information.
- *Element vs. structure matching*: Element matchers consider matching between atomic schema elements such as attributes whereas structure-level matchers can deal with combinations of elements, e.g., by comparing sets of attributes of two classes.
- *Language vs. constraints*: Language-based matchers use textual information and linguistic techniques for matching. Examples are equality or similarity of element names as well as a thesauri-based identification of synonyms and hypernyms. A second approach is to consider constraints defined as part of the schema, e.g., data types, cardinalities of relationships or key characteristics.
- *Matching cardinality*: Another kind of characterization is the cardinality of matches. For example, an 1:1 match means that an attribute for one schema is mapped to another attribute of the second schema. An 1:*n* mapping means that a single attribute is mapped to a set of other attributes, e.g., by computing a value from the other values or the extension of one class is computed by combining the instances from several other classes of the second schema.
- *Auxiliary information*: Often external information can be used to support the identification of matches. This can be provided in the form of user input, results from previous steps or by using thesauri, dictionaries, ontologies etc.

In [3] several available systems are presented and compared based on the support of the classification criteria described above. Other example systems are, e.g., Rondo [6] or ToMAS [7]. However, only two of the seven systems considered in [3] are rule-based and one of them requires to implement the rules in Java. Thus, extending or adapting the matcher requires often a high coding effort which could be reduced by providing a set of built-in predicates representing individual matchers. Based on these predicates the user could define composite matchers in the form of rules which could be used for deriving the mapping information, too.

3. Problem statement

In this paper, we consider schema matching in the context of semantic integration systems. Here, the matching has to be performed mostly between the global "semantic" schema (the domain knowledge model or the ontology) and the local source schemas. Therefore, matching between schema elements often cannot be expressed based on simple name or structural matching. Instead, domain knowledge in the form of constraints, relationships, thesauri etc. has to be taken into account. In order to provide a better understanding of the specialities as well as the potential of (extensible) schema matching in ontology-based mediators we first introduce the integration model of our mediator system YACOB [8] and present the mapping specifications necessary for registering new sources.

YACOB is a mediator system developed for the integration and querying of Web databases on cultural assets that were lost or stolen during World War II, such as www.lostart.de. In order to capture semantically rich information, a two-level model is used for integration:

- the meta or concept level describing the semantics of the data and their relationships as well as
- the actual data or instance level representing the data provided by the sources.

The model layer for representing concepts is based on RDF Schema. The Resource Description Framework (RDF) developed by the W3C describes a simple graph-based model consisting of nodes, which model resources (e.g., Web documents) and literals, and edges representing properties of resources. RDF Schema (RDFS) extends this model by introducing primitives like classes and class relationships which are useful for specifying vocabularies or ontologies. RDFS is similar to traditional (object-oriented) database models, but contains some special features, e.g., properties are defined independently from classes and are restricted in their association with classes only by specifying domain and range constraints.

In our integration model we treat classes as so-called concepts and add a second kind of class: categories. The difference between these two kinds of classes is as follows: a concept is a class for which extensions (data objects) are provided by the sources. In contrast, categories are abstract classes without extensions representing abstract property values. They are used to capture terms represented in different sources by different values. Furthermore, RDFS properties correspond to concept properties in our model. Relationships

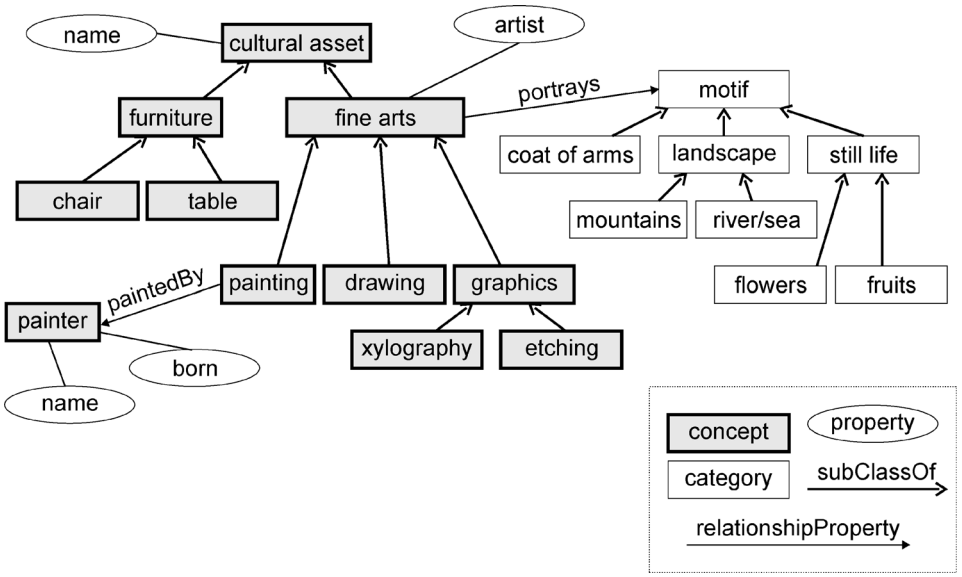

Fig. 1. An example ontology.

between concepts are also modeled as properties where the domain and range are restricted by concepts.

Fig. 1 shows an example of a concept schema modeled using these elements. This schema defines a hierarchy of concepts representing different kinds of cultural assets. In this example only few properties are shown. An example of the usage of categories is the property “portrays”. The domain of this property is the category “motif” for which additional sub-categories exist. Even if these categories are represented by different property values in the source systems, at the global level we can always refer to the globally defined terms. Another example is given for relational properties: the property “paintedBy” relates paintings and painters.

At the instance level data is represented in XML both inside the mediator (i.e., for transformation and query processing) as well as during exchange between the mediator and the sources. For the sources, we assume they are able to export data (query results) in XML structured according to a (nearly) arbitrary DTD and can answer simple XPath queries. In case of necessary transformation for XPath to the source’s query interface wrappers are required. Because we allow arbitrary DTDs for data exchange the transformation into the global schema (defined by the concept schema) is performed by the mediator.

We can now define a concept schema by sets of facts and rules as shown in the following example:

```
concept(CulturalAsset).
concept(FineArts).
sub-concept(CulturalAsset, FineArts).
property(CulturalAsset, Name).
...
```

```
<results>
  <paintings>
    <title>Mary with the child</title>
    <person>Gossaert, Jan</person>
    <material>Eichenholz</material>
    <url>http://www.lostart.de/recherche/einzelobjekt.php3?
         lang=english&einzel_id=7049</url>
    <image_url>
         http://www.lostart.de/recherche/bild.php3?id=7836
    </image_url>
  </paintings>

</results>
```

Fig. 2. Sample data from a source.

```
sub-concept(x, z) :- sub-concept(x, y), sub-concept(y, z).
property(x, p) :- sub-concept(y, x), property(y, p).
```

In a similar way, we could represent data in a logic-based form. However, because in this paper we focus on schema level matching, we do not consider this here. Although inside the mediator schemas and data are represented as RDF and XML data, this can always directly be converted into an equivalent logical representation. An example of an XML data representation returned by a source query is shown in Fig. 2.

Besides features for specifying schemas and representing data a mediator system requires a specification mechanism for mappings or correspondences. One approach for correspondence specification is the definition of views expressed in a query language. Another approach which we have chosen is to represent correspondences in the form of properties associated with elements of the concept schema. In this way, a mapping describing how a source provides data for a certain concept is defined by annotating concepts and properties of the concept schema.

In our model, we distinguish between the following kinds of mappings:

- Concept mappings specify how the given global concept c is supported by a given source. A concept mapping comprises the following information:
  - The source name for identifying the source from where the instances are to be retrieved.
  - The name of the local XML element used for representing instances in the source.
  - An optional filter predicate for further restricting the instance set, e.g., to address only instances satisfying a certain condition.

  $$c \mapsto \langle src, path\text{-}to\text{-}elem, filter \rangle.$$

- A property mapping defines the correspondence between the property p of a concept and an XML element or attribute of the source data. This can be represented by giving the source name and a path expression to the XML element:

  $$p \mapsto \langle src, path\text{-}to\text{-}elem \rangle.$$

```
<yacob:ConceptMapping rdf:about="Lostart_Painting">
  <yacob:sourceName>lostart.de</yacob:sourceName>
  <yacob:localElement>/results/paintings</yacob:localElement>
</yacob:ConceptMapping>

<rdf:Description rdf:about="painting">
  <yacob:providedBy rdf:about="Lostart_Painting">
</rdf:Description>

<yacob:PropertyMapping rdf:about="Lostart_name">
  <yacob:sourceName>lostart.de</yacob:sourceName>
  <yacob:pathToElement>/results/paintings/title</yacob:pathToElement>
</yacob:PropertyMapping>
```

Fig. 3. Sample mapping specification.

- Join mappings are required if a property represents inter-source relationships (e.g., *paintedBy* in Fig. 1), i.e. where the related concepts are supported by different sources. In this case, a traversal of this relationship has to be translated into a join operation between the concept extensions. However, this kind of mapping affects only the concept level without referring to the actual source schemas. It is used for query rewriting only and has not be considered during schema matching. Therefore, we do not consider it here.
- Value mappings are used for defining how a category term is mapped to a literal value in a source. For this purpose, the source name and the literal are required only:

$$v \mapsto \langle src, literal \rangle.$$

Based on these mappings correspondences are specified for each source separately in a GLAV style: concept, property and value mappings are LAV, whereas join mappings are in fact global views which means a GAV approach. In this way, source mapping specifications are independently from each other and a matching is necessary only between each individual source and the global ontology.

Identifying matchings and defining mappings of these kinds are the main task of schema mapping in the YACOB system. To each concept supported by a given source, appropriate concept mappings and the accompanying property and value mappings are assigned. However, due to the existence of specialization relationships between concepts not every concept has to be annotated. Instead, only concepts which represent a leaf in the hierarchy with respect to a given source have to be considered. An example of a mapping specification is given in Fig. 3 where `providedBy` associates the mapping to the concept `painting` and `sourceName` corresponds to *src* and `localElement` to *path-to-elem* in $c \mapsto \langle src, path\text{-}to\text{-}elem, filter \rangle$.

The mapping specifications are used both for result transformation as well as query translation. Query results are transformed by applying source-specific XSLT rules which can be automatically derived using the following rules:

(1) For a concept mapping to $c \mapsto \langle src, lelem \rangle$, the following XSLT template is generated:

```
<xsl:template match="lelem">
  <c> <xsl:apply-templates /> </c>
</xsl:template>
```

(2) For a property mapping $p \mapsto \langle src, lelem1/lelem2 \rangle$, a corresponding XSLT template of the following form is derived:

```
<xsl:template match="lelem1">
  <p>
    <xsl:value-of select="lelem2" />
  </p>
</xsl:template>
```

(3) For a property p with a domain consisting of the hierarchy of categories $k_1, k_2, \ldots, k_m$ with associated value mappings $v_1 \mapsto \langle src, val_1 \rangle \ldots v_m \mapsto \langle src, val_m \rangle$, the following XSLT template for the property mapping is created:

```
<xsl:template match="elem1">
  <p> <xsl:choose>
    <xsl:when test="elem2/elem3 = 'v1'">
      <xsl:text>k1</xsl:text>
    </xsl:when>
    <xsl:when test="elem2/elem3='v2'">
      <xsl:text>k2</xsl:text>
    </xsl:when>
    ...
  </xsl:choose> </p>
</xsl:template>
```

Please note that we do not need an XSLT template for join mappings because such properties are handled during query rewriting.

Query translation is performed by deriving an expression in an extended query algebra which provides additional operators for dealing with concept level operations (e.g., set operators, path traversal including transitive closure, etc.) as well as an operator obtaining the extension of a given concept. In the next step, intersource relationships are substituted by join operations using the join mappings. Then, the query is processed by first evaluating the concept-level operators. Here, several heuristics are applied taking constraints from the ontology level (specialization relationships) into account. Finally, the remaining subqueries are translated into source queries on the basis of the concept and property mappings. For further details on query rewriting and processing we refer to [8].

In the following sections we will discuss how the process of matching and mapping derivation can be supported by existing schema matching approaches and how these approaches can be improved towards an extensible approach.

4. Rule-based schema matching

Considering existing schema matching approaches (cf. Section 2) we have a wide variety on different matching criteria at hand leading to quite different schema matchers (i.e.,

schema-based, instance-based, language-based, ...). Although there are already proposals for combining these schema matchers and building so-called hybrid matchers, such combinations are typically static and rather inflexible.

Here, employing logical mechanisms promises significant benefits in particular concerning flexibility in combining different schema matchers and expressing a large spectrum of matching criteria within one framework. In the following we therefore discuss which kinds of properties and criteria we want to express and present the logical concepts forming a rule-based general framework for schema matching.

Such a logical framework offers several advantages like:

- extensible matchers;
- proper means to deal with semantically rich schemas (ontologies);
- (semi-)automated derivation/refinement of matching rules based on already mapped sources (e.g., identifying synonyms, constraints, etc.)

In the following subsections we introduce rule-based concepts for schema matching step by step. First we briefly discuss how uncertainty can be represented in logical languages. Then, we show how in general rules for elementary matchers, the property matchers, look like. Thereafter, we investigate the combination of property matchers in order to obtain rule sets for concept matching. Finally, we discuss the evaluation of matching rules stating on which basic principles the evaluation can be realized.

4.1. Representing uncertainty

First of all we need to represent uncertainty because even elementary schema matching algorithms (for individual matchers) often do not produce crisp results (cf. [9]). There is in general an inherent uncertainty due to the assumptions the schema matching algorithm relies on, due to missing semantic information, or due to the algorithm itself (e.g., for algorithms based on statistic analysis techniques).

There are several approaches to incorporate uncertainty into logical languages. Most of these approaches represent uncertainty as probabilities, e.g. [10,11]. For our purposes a probabilistic extension of Datalog as introduced in [11] (for which an algebraic semantics was given in [12]) seems to be most appropriate. In fact, for this particular approach there is some first work on its usage for schema matching [13].

Adding probabilities (or uncertainty values) to a logical language can mainly be done on two levels:

- On the object level we can decorate facts with an information on their certainty or uncertainty. In this way we are able to express that a fact is not necessarily true or false, but that there is, e.g., a certain probability for this fact to hold.
- On the rule level we can decorate rules with an information on their certainty or uncertainty. Not only facts can be uncertain but also rules.

In the context of schema matching we usually have uncertain facts, e.g. expressing that an object class in one source corresponds to an object class in another source with some uncer-

tainty. But uncertain rules might occur as well. We can think of producing a set of matching rules for a concrete application. These rules can then be used for matching concrete instances which is in particular useful and necessary if the databases (or sets of documents) are modified. Here, the schema matching is usually done once producing a set of matching rules for the data related to the schemas matched. Due to the fact that for schema matching there is only a restricted amount of (semantic) information available the result will in general not be perfect, i.e. there will always be some uncertainty about the exact matching on the instance level.

Using probabilities as representation for uncertainty has the advantage that the rules for computation with probabilities are well understood and easy to implement. Although the discussion about which model for representing uncertainty is most adequate is beyond the scope of this paper, we have to point out that other models could be used as well. Looking at data mining algorithms which might be used to produce some input for a schema matching algorithm we have to face the fact that some algorithms yield probabilities while others yield possibilities, support and confidence values, or other representations for uncertainty as result. At that level we obviously might run into another integration problem if we have to deal with heterogeneous representations of uncertainty.

For the purposes of this paper we do not need to care about the concrete representation model for uncertainty. For simplicity, we use probabilities having in mind that other representation models for uncertainty might need other rules for computation. For implementing a rule based schema matching within a concrete application scenario we then have to decide about the representation model.

4.2. Rules for property matching

As a starting point (in a bottom-up view of the matching process) we need matching rules which allow to match with respect to specific properties. These rules may use built-in predicates implementing such specific atomic matchers. For instance, simple matching criteria like matching strings (on schema level identifiers for classes, attributes, etc.) based on common prefixes could be provided as built-in predicate. Looking at this example of a prefix matcher, there is a built-in predicate `prefix_match` having (at least) three arguments: two arguments for the strings/identifiers to be matched and one argument (called `conf` here) for a value expressing how good the match is:

```
match(p1, p2, conf) :- prefix_match(p1.name, p2.name, conf).
```

This rule simply expresses that two structured objects (e.g., XML documents) match as well as a prefix match on their names. In the same style it is possible to use other property matchers, for instance a matcher using the edit distance (e.g., as built-in predicate `edistance`):

```
match(p1, p2, conf) :- edistance(p1.name, p2.name, dist),
                       conf = 1.0 - dist /
                            max(len(p1.name), len(p2.name)).
```

For our example from Section 3 (cf. Figs. 1 and 2) we could try to match, e.g., the name of the concept `painting` with the name of the XML element `paintings` which would result in a rather high value for `conf` in the `match` predicate. If we try to match `furniture` with `image_url` this will produce a very small confidence value for this match.

Furthermore, property matchers can be combined by checking several different features, e.g., the edit distance and substring containment, and derive and matching similarity (here, simply the average):

```
match(p1, p2, conf) :- edistance(p1.name, p2.name, dist),
                       conf1 = 1.0 - dist /
                            max(len(p1.name), len(p2.name)),
                       substring(p1.name, p2.name, len),
                       conf2 = len /
                               min(len(p1.name), len(p2.name)),
                       conf = (conf1 + conf2) / 2.
```

Of course, there is no problem to also have crisp criteria for matchers like equality (e.g., assuming that the argument `conf` may have values in the range from 0.0 to 1.0):

```
match(p1, p2, 1.0) :- equal(p1.name, p2.name).
match(p1, p2, 0.0) :- not-equal(p1.name, p2.name).
```

4.3. Rules for concept matching (combining property matchers)

In the previous section we introduced rules based on property matchers. Of course, property matchers used in isolation are in general not able to yield an adequate result. The result of a single property match needs to be considered within the context of the properties compared. For that, other classes and their properties in the direct neighborhood need to be compared. As a general principle we can state that the larger the portion of the neighborhood is which can be matched as well, the more adequate the match is.

For going into details we consider the ontology given in Fig. 1 and the XML fragment depicted in Fig. 2.

Property matchers can be employed for finding candidates representing the same semantic information in the ontology and in the XML document. Trying to match names of object classes or properties in the ontology with tag names in the XML document will show that for instance `painting` (concept in the ontology) and `paintings` (tag in the XML document) are candidates to represent the same concept in the real world. Without any additional information this does not really help. In other examples we might find several conflicting pairs of such candidates. On the other hand, property matchers can often not detect the right candidates, for instance it is not probable that a simple property matcher finds out that `person` and `artist` represent the same semantic concept. In this case, thesauri or dictionaries are helpful which can be incorporated by an appropriate rule:

```
/* thesauri */
synonym('person', 'artist').
...
```

```
match(c1, c2, conf) :- edistance(c1.name, c2.name, dist),
                       conf1 = 1.0 - dist /
                             max(len(c1.name), len(c2.name)),
                       match_children(c1, c2, conf2),
                       conf = f(conf1, conf2).

match_children(c1, c2, cavg) :-
                 cavg = avg(conf, (child(c1, cc1),
                            child(c2, cc2),
                            best_match(cc1, cc2, conf)).

best_match(c1, c2, cmax) :- cmax = max(conf,
                                       match(c1, c2, conf)).
```

Fig. 4. Rules for matching hierarchical structures (simplified version).

```
match(p1, p2, conf) :- synonym(p1.name, s),
                       equal(s, p2.name),...
```

Obviously, we should try to use several different property matchers and combine their results. In particular, using the graph (or tree) structures of the ontology and of the XML document allows a semantics driven procedure to find adequate global matches. For this, XML documents are taken as trees where a tag represents a node in the tree (the node is labeled by that tag). Child nodes can directly be accessed from a node by using their tags.

Now, having a concept in the ontology and a node (tag) in the XML document as candidates for representing the same semantic information, we can compare their properties or sub-concepts and child nodes, respectively, to see whether they can be matched as well. This can and has to be done recursively along the graph or tree structure. Because these comparisons for finding matches in sub-structures are usually not equally important for assessing the quality of a possible match, we can add weights. For instance, weights can be used to express that matches of direct sub-structures (properties and child nodes) contribute more to the certainty of a specific match than matches of sub-structures with a larger distance to the considered concept and/or XML node.

In a bottom-up evaluation we first have to find matching candidates for the leaves of a XML document tree (or of the concept graph). For this we can employ property matchers. Then, we can combine their results for those leaves being child nodes of some inner node. For that, we have to provide combination rules. The step of combining results of sub-trees has to be repeated until we reach the root of the document tree. Fig. 4 depicts logical rules which provide the principal structure for that (where `f` stands for a function computing a weighted confidence).

Due to the fact that there are often several possible matches for a concept or node, we have to eliminate unlikely matches. One possible solution is to simply choose the best match. For combining the `conf` values for all children we have to specify a corresponding function. Here, we decided to simply use the average. Clearly, applying such aggregation functions we go beyond the expressiveness of first order logic. As we explain in Section 4.4 this does not raise severe problems because a safe evaluation can be achieved.

Of course, these logical rules only represent a very simplified matching model. There are several aspects requiring more complicated matching rules here:

- The `match_children` rule does not really provide a reasonable result. This is due to the fact that this rule as it is compares each child of c1 with each child of c2, determines a best match for this pair of children and then computes the average over all pairs.
  For a realistic usage we have to find for the *n* children of c1 and the *m* children of c2 a (partial) mapping which tells us for which children of c1 there are corresponding children of c2. For these pairs of corresponding children we can then compute the best match before we combine all these values to one value for the match between c1 and c2.
  In our example (cf. Figs. 1 and 2) we may want to match `painting` (a concept in the ontology) with `paintings` (an element in the XML document). `painting` has three properties:
  – `name` (inherited from `cultural asset`),
  – `artist` (inherited from `fine arts`), and
  – `painted by`.
  In the XML structure `paintings` has five properties (i.e. sub-elements):
  – `title`
  – `person`
  – `material`
  – `url`
  – `image_url`
  Here, the `match_children` rule would investigate all 15 combinations (different pairs consisting of one property of `painting` and one property of `paintings`), find the best match confidence for each of the 15 pairs, and compute the average value out of the 15 match confidence values. Comparing two concepts or elements with a large number of properties having no counterpart (which should result in match values equal or close to 0), the average value will be very low.
  Ideally, properties or sub-concepts with no counterpart in a match should not contribute to the final confidence value (or at most in a very limited way). In our example, for the three properties of `painting` there should be at most three properties of `paintings` as counterparts. It is clear, that at least two properties of `paintings` cannot really provide to the confidence for the match of `painting` and `paintings`.
- Another problem arising in the example is that sub-concepts or properties, which should be compared for coming out with a reasonable result, are not always on the same level. For instance, for matching `painting` in the ontology with `paintings` in the XML structure the name of the painter is only an indirect property of `painting` (via the relationship property `paintedBy` to the concept `painter`) whereas the name of the painter is directly given by the direct sub-element `person` for `paintings`.
  Therefore, a more sophisticated matching model should take different “distances” for possibly matching sub-concepts or sub-properties into account.
- Here, we treat all children of a node in the same way using the predefined predicate `child`. Having different kinds of child nodes we could use different predicates to distinguish them. For instance, for ontologies we may want to distinguish between properties of a concept and sub-concepts of that concept.

Instead of the predicate `child` we could introduce a set-valued function for returning the set of all child nodes. Set-valued or more general multi-valued functions (returning sets, bags, or lists of values or objects) could in addition be used for expressing aggregation in a much more simpler way.

4.4. Evaluation of matching rules

As a basis for our logical framework we could use first-order logic. The syntax we used here for examples of rules has been adopted from Datalog [14]. Obviously, basic (positive) Datalog is not sufficient. In the previous sections we pointed out a number of further concepts like negation, complex objects (i.e., objects with sub-objects, set-valued attributes, etc.), aggregation, etc. There is a whole bunch of work on extending Datalog by such concepts. For instance, LDL [15] adds sets and negation to Datalog. For reasoning about complex (structured) objects a number of rule-based approaches are available (e.g., [16–18]).

Such extensions allow us to investigate the child nodes of a node (i.e., tags enclosed within another tag) in such a logical framework by accessing the features (properties/attributes) of an object. All nodes (e.g., corresponding to elements in XML documents) are considered to be objects, the child relationship is expressed by properties of the objects, where the corresponding tags from the XML document provide to the names of the properties.

As basic evaluation concept we employ stratification. Stratification is a common bottom-up evaluation mechanism originally developed for capturing negation in deductive languages (see for instance in [19]). The basic concept of stratification has then been applied as an evaluation technique for quite a number of other additional concepts added to a deductive language. The deductive language StateLog which allows explicit references to past states in the evaluation process [20] is a nice example for a sophisticated employment of stratification.

For evaluating matching rules in our setting we particularly need to get a grasp of aggregation operations. For expressing different matching strategies we need for instance to be able to say that we are only interested in the best match (i.e., the match with the maximum matching confidence). To find the maximum we first have to compute all possible matches with their matching confidence before we can continue with the best match. Obviously, a stratified evaluation of the corresponding rules can solve this in a natural way (a detailed discussion on aggregation in deductive languages can be found in [21]).

4.5. Process of rule-based matching

The integration of a new source requires to import the local source schema and to apply the matching rules. Because in most cases an automatic matching is difficult we have to deal with several possible candidate matches. From this candidate set the schema integrator can choose an appropriate match and derive the corresponding mapping specification. After the mappings for all relevant elements from the local schema are specified and imported into the mediator the source can be queried and accessed from the global level.

```
procedure find-matches (concept c, concept e, confidence conf, candidates cset)
    evaluate best-match (c, e, nconf)
    if nconf ⩾ conf then
        cset := cset ∪ (c, nconf)
        foreach c' ∈ subconcepts(c) do
            find-matches (c', e, nconf, cset)
        od
    else
        /* no better matching possible—stop the evaluation */
    fi
end

algorithm schema-match (schema s, concept hierarchy C)
    c := root(C)
    foreach schema element e ∈ s do
        cset := ∅

        find-matches (c, e, 0, cset)
        clist := sort_conf (cset)
        output clist
    od
end
```

Fig. 5. Algorithm for rule-based schema matching.

In order to apply the matching rules it is necessary to represent the local schema in the same model as the concept schema. For our case of XML sources this is achieved by treating the elements of the XML DTD as concepts and attributes or sub-elements as properties. If no DTD is available, it can be inferred from the actual XML data.

The next step is to process the matching rules. Here, a first possible approach is a strict logic-based bottom-up evaluation. Starting with the evaluation of property matching rules this approach combines these matchers using the concept matching rules in order to determine the best match. Depending on the strategy only the best match (i.e., with the highest confidence value) or a set of candidate matches (i.e., with a confidence value greater than a given threshold) are returned. However, this approach requires to check the match of each combination of source and global properties and therefore requires a high effort. In addition, specialization relationships between concepts are not exploited.

An alternative approach is to evaluate the matching rules along the specialization hierarchy of the concept schema. This combines the bottom-up evaluation of rules with a graph-based top-down approach. The algorithm as shown in Fig. 5 works as follows.

For each concept e representing a source schema element we start with the root concept(s) of the ontology C. Using the concept matching rules we check by bottom-up evaluation if there exists a match, i.e. if $conf > 0$. By proceeding downwards, i.e. following the specialization relationships, and applying the matching rules again, we check if the match is improved, i.e., if the new confidence value $nconf \geqslant conf$. In this case, we can proceed recursively at the next level etc. If the confidence value decreases we can stop the evaluation for the remaining sub-tree. This could arise, if for a sub-concept some additional properties are defined for which no corresponding properties exist in the source schema concept. In this way, we can guarantee that we found a match for the most specific

concept in the hierarchy without considering each concept of the schema. After the hierarchy traversal has been finished we collect a set of candidate matches *cset*. This set can be ordered by the confidence value (*clist*) and presented to the user.

The final step is to derive the mapping information as introduced in Section 3 required for transforming global queries as well as result data. Basically, these mappings are implicitly given by the predicates and variable substitutions used for evaluating the matching rules. This means for example, if a match was chosen between the XML element `paintings` (containing the sub-elements `artist_name` and `title`) and the concept `painting` (with properties `artist` and `title`) due to the following rule evaluation

```
match(c, xml, conf)
 edistance("painting", "paintings", 1)
 match_children(c, xml, conf2)
  equal("title", "title")
  prefix_match("artist", "artist_name", conf3)
...
```

we can directly derive the following mappings for a source *src*

$$\begin{aligned} \mathit{painting} &\mapsto \langle \mathit{src}, \mathit{paintings}, \text{true} \rangle \\ \mathit{artist} &\mapsto \langle \mathit{src}, \mathit{paintings}/\mathit{artist\_name} \rangle \\ \mathit{title} &\mapsto \langle \mathit{src}, \mathit{paintings}/\mathit{title} \rangle \end{aligned}$$

One way to derive these mappings for a complete matching step is to use the evaluation trace of the rule engine. If we are able to determine which predicates and rules were used we can derive the mapping specification.

Another approach is to explicitly encode the mapping generation as part of the matching rules. This can be simply done by introducing an additional variable representing the mapping string which is completed in each property and concept matching rule as sketched in the following example:

```
match(c1, c2, conf, mapping) :-
                    edistance(c1.name, c2.name, dist),
                    concat(mapping, "c1 ↦ ⟨src, c2.name, true⟩"),
                    ...
                    match_children(c1, c2, conf2, mapping),
                    ...

match(p1, p2, cname, conf, mapping) :-
                    edistance(p1.name, p2.name, dist),
                    concat(mapping, "p1 ↦ ⟨src, cname/p2.name⟩"),
                    ...
```

In this way, after the matching of a concept-level rule the whole mapping specification is available.

5. Conclusions

An important task of integrating data from heterogeneous sources is the problem of schema matching in order to be able to derive mapping information required for query rewriting and translation as well as result data transformation. Though several approaches were proposed aiming to support a (semi-)automatic matching, there is still a need for extensible solutions allowing to add application-specific or domain-specific matching criteria. In this paper, we have discussed a logic-based framework using rules for schema matching and offering a high degree of flexibility in combining different matchers in a domain-specific way. We have further shown that such a logical representation is particularly useful in scenarios where the global schema is represented in the form of an ontology supporting the modeling of different kinds of relationships and in this way hiding semantic correspondences. In the first step, we have only addressed the problem of schema-level matching but we are aware of the need to consider instance-level information, too.

References

[1] M. Lenzerini, Data integration: a theoretical perspective, in: Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Databases (PODS 2002), 2002, pp. 233–246.
[2] M. Friedman, A. Levy, T. Millstein, Navigational plans for data integration, in: AAAI/IAAI 1999, 1999, pp. 67–73.
[3] E. Rahm, P. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (4) (2001) 334–350.
[4] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, in: Int. Conference on Very Large Data Bases (VLDB) 98, 1998, pp. 122–133.
[5] A. Doan, P. Domingos, A. Halevy, Reconciling schemas of disparate data sources: a machine-learning approach, in: SIGMOD Conference 2001, 2001, pp. 509–520.
[6] S. Melnik, E. Rahm, P.A. Bernstein, Rondo: a programming platform for generic model management, in: SIGMOD Conference 2003, 2003, pp. 193–204.
[7] Y. Velegrakis, R. Miller, L. Popa, Mapping adaptation under evolving schemas, in: Int. Conference on Very Large Data Bases (VLDB) 2003, 2003, pp. 584–595.
[8] K. Sattler, I. Geist, E. Schallehn, Concept-based querying in mediator systems, VLDB J. (2004), in press.
[9] E. Altareva, S. Conrad, Statistical analysis as methodological framework for data(base) integration, in: I.-Y. Song, S. Liddle, T. Ling, P. Scheuermann (Eds.), Conceptual Modeling—Proceedings of the ER'2003 Conference, Oct. 2003, in: Lecture Notes in Computer Science, vol. 2813, Springer-Verlag, 2003, pp. 17–30.
[10] J. Heinsohn, A probabilistic extension for terminological logics, in: B. Nebel, K.V. Luck, C. Peltason (Eds.), Proceedings of the International Workshop on Terminological Logics no. D-91-13 in DFKI-Report, DFKI, 1991, pp. 51–55.
[11] N. Fuhr, Probabilistic Datalog—a logic for powerful retrieval methods, in: E. Fox, P. Ingwersen, R. Fidel (Eds.), SIGIR'95, Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR Forum, ACM Press, 1995, pp. 282–290.
[12] N. Fuhr, T. Rölleke, A probabilistic relational algebra for the integration of information retrieval and database systems, ACM Trans. Inform. Syst. 15 (1) (1997) 32–66.
[13] H. Nottelmann, N. Fuhr, Combining DAML + OIL, XSLT and probabilistic logics for uncertain schema mappings in MIND, in: T. Koch, I. Solvberg (Eds.), European Conference on Digital Libraries (ECDL 2003), in: Lecture Notes in Computer Science, vol. 2769, Springer, 2003.
[14] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about Datalog (and never dared to ask), Knowledge Data Engrg. 1 (1) (1989) 146–166.
[15] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmuelli, S. Tsur, Sets and negation in a Logic Database Language (LDL1), in: ACM SIGMOD Conference on Principles of Database Systems (PODS'87), 1987, pp. 21–37.

[16] A. Heuer, P. Sander, Semantics and evaluation of rules over complex objects, in: W. Kim, J.-M. Nicolas, S. Nishio (Eds.), Deductive and Object-Oriented Databases, Proc. First International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Dec. 1989, North-Holland/Elsevier, 1990, pp. 473–492.
[17] S. Abiteboul, S. Grumbach, COL: a logic-based language for complex objects, in: J. Schmidt, S. Ceri, M. Missikoff (Eds.), Advances in Database Technology—EDBT'88, Proc. International Conference on Extending Database Technology, Venice, Italy, 1988, in: Lecture Notes in Computer Science, vol. 303, Springer-Verlag, 1988, pp. 271–293.
[18] M. Kifer, G. Lausen, F-Logic: a higher-order language for reasoning about objects, inheritance, and scheme, in: J. Clifford, B. Lindsay, D. Mayer (Eds.), Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Oregon, in: ACM SIGMOD Record, vol. 18(2), ACM Press, 1989, pp. 134–146.
[19] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer-Verlag, Berlin/New York, 1990.
[20] G. Lausen, B. Ludäscher, W. May, On logical foundations of active databases, in: J. Chomicki, G. Saake (Eds.), Logics for Databases and Information Systems, Kluwer Academic Publishers, Boston, 1998, Ch. 12, pp. 389–422.
[21] C. Zaniolo, Key constraints and monotonic aggregates in deductive databases, in: A.C. Kakas, F. Sadri (Eds.), Computational Logic: Logic Programming and Beyond—Essays in Honour of Robert A. Kowalski, in: Lecture Notes in Computer Science, vol. 2408, Springer, 2002, pp. 109–134.