Rewriting Queries by Means of Model Transformations from SPARQL to OQL and Vice-Versa

Guillaume Hillairet, Frédéric Bertrand, and Jean Yves Lafaye

Laboratoire Informatique Image Interaction,
University of La Rochelle, France
{guillaume.hillairet01,fbertran,jylafaye}@univ-lr.fr

Abstract. Implementing language translation is one of the main topics within the model to model transformation area. Nevertheless, a majority of solutions promoted by model driven engineering tools focus on transformations related to modeling languages. In this paper, we address query rewriting by means of model transformations. This study has been carried out within the context of implementing an object ontology mapping tool, which could enable bridging object oriented applications and RDF data sources. This approach allows querying RDF data sources *via* an object oriented query which is automatically rewritten in SPARQL (RDF query language) in order to access RDF data. Hence, the developer can freely focus upon the sole application object model. In this paper, we also present with solutions for translating SPARQL queries into object oriented queries, thus allowing the implementation of SPARQL endpoints for object oriented applications.

Keywords: Model Transformations, Query languages, SPARQL, OQL, ATL.

1 Introduction

The Semantic Web [4] envisions the promotion of the Web of Data as a giant interlinked information database [6]; data is expressed and published in a machine-readable format, which enables automatic processing and inference. Semantic Web technologies provide three facilities to manage data, namely: a directed labeled graph model for data representation: RDF [12] (Resource Description Framework); a Web ontology language (OWL [3]) to express data semantics and a query language for RDF (SPARQL [16]) which allows to distribute queries across RDF graphs. These technologies provide a pragmatic way to make the Semantic Web vision operational. Integrating these technologies within an application development process may be valuable in two ways: firstly, Web applications may provide their data in RDF and thus contribute to Web of Data enrichment; secondly, applications could use any available RDF data on the Web and thus enrich their own content (ex: automated data mashup [1]).

One way to achieve this is to allow the manipulation of RDF data as plain objects, thanks to an object/ontology mapping solution, similar to what exists for bridging relational data and object oriented applications. One main common feature of such tools is their dealing with query rewriting. During the implementation of an object

R. Paige (Ed.): ICMT 2009, LNCS 5563, pp. 116–131, 2009.

ontology mapping, we actually faced the problem of designing a query rewriting tool. This tool should allow the developer to successively write a query according to the application object model, rewrite this object oriented query as a SPARQL query thanks to the previously defined mapping between the object model and an ontology, execute the query on a RDF data source, and translate back the results into objects.

In this paper we study the transformation rules between an object oriented query language and the RDF query language SPARQL. We develop a query rewriting engine on top of an object ontology mapping solution. The query rewriting engine uses the ATL model transformation language [14]. This study demonstrates a novel application area for model transformations.

The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 introduces the context of this work, which is based on the implementation of an object ontology mapping solution performing RDF data access through an object abstraction. Section 4 presents the query languages we chose for this study. Section 5 explains the query rewriting process implementation *via* model transformations. Finally, Section 6 concludes on remarks and future works.

2 Related Work

Providing solutions to expose existing data source content in RDF is of high interest for insuring a real adoption of the Semantic Web. Many approaches have been proposed, mainly about relational to RDF mapping. The W3C RDB2RDF Incubator Group produced a corresponding typology. Other states of the art for database to RDF efforts can be found in [9] and [19]. Some of these approaches provide mechanisms to query relational data with SPARQL.

R2O [18] and D2RQ [5] are based on declarative mapping languages, and can be used to build SPARQL endpoints over RDBMs. R2O provides more flexibility and expressiveness, it needs a referenced ontology. D2RQ directly exposes the database as Linked Data and SPARQL with D2R server. Using either R2O or D2RQ requires an initial learning of the language as well as a good knowledge about modeling. Virtuoso RDF Views [8] stores mappings in a quad storage. While D2RQ and RDF Views follow the table-to-class, column-to-predicate approach, RDF Views has some additional methods to incorporate the DB semantics. All mapping languages have more or less the same complexity and flexibility as SQL. Relational.OWL [11], SPASQL [17] and DB2OWL [9] are other projects that aim to expose relational data on the Web.

SPOON (Sparql to Object Oriented eNgine) [7] addresses the wrapping of heterogeneous data sources in order to build a SPARQL endpoint. The SPOON approach grounds on an object oriented abstract view of the specific source format (as in ORM solutions). SPOON provides a run time translation of SPARQL queries into an OO query language based on the correspondence between the SPARQL algebra and the monoïd comprehension calculus. Our approach shares many conceptual ideas with the SPOON project. Nevertheless, the SPOON approach seems to only operate in case of an ontology that is generated from an object model, and not to offer solutions for complex mappings. Furthermore, SPOON does not propose any solution for rewriting OO queries into SPARQL queries.

3 Context of This Work

In this section, we present the context in which our proposal has been developed. The query rewriting process uses an object ontology mapping solution we are currently developing. This solution allows both RDF data access through an objet abstraction, and conversely ensures an external access to objects *via* SPARQL queries on a corresponding RDF representation. It does alleviate the implementation of SPARQL endpoints.

3.1 Bridging Object Oriented Applications and Semantic Web

The pieces of work presented in this article take place within the context of the development of an object ontology mapping solution. We already presented our main ideas in [10]. We have specified a declarative mapping language and developed a framework that is inspired from object relational mapping framework, but actually deals with bridging applications and Semantic Web sources.

Such a solution allows the publication of data being generated by an object oriented application (here, by a Java application) as RDF data. RDF is dynamically generated and can be used by external applications, taking advantage of the data integration capabilities offered by the Semantic Web technologies. This allows an easier data sharing between applications, for example automated data mashups. The second benefit of such a solution is to allow the developer focusing on the application object model, without having to care about how objects are represented in RDF, and thus not having to define SPARQL queries, but rely upon an object query language instead. One advantage is that the application is no longer bound to a single data source, as usual in object relational solutions.

The framework we developed considers the set of POJO (Plain Old Java Object) classes as the domain object model. POJO classes are represented at runtime as an Ecore model (i.e. a metamodel), thus enabling the representation of object graphs as models conforming to an Ecore model. Considering the application domain model as a metamodel is perhaps unusual, but not penalizing, since classes of the application domain model are supposed to be POJO classes. Such classes generally own properties that can be accessed by getters and setters; Class operations can specify the class behavior. This kind of classes can easily be represented in the form of Ecore models. The use of model enables the execution of model transformations at runtime for both data transformation and query rewriting. Our framework uses the ATL model transformation language.

The object-oriented model proposed by Ecore [20] and the ontology model proposed by OWL [3] share many features. Both include the concepts of classes, properties and multiple inheritances. However, the object model and the ontological model show major differences as reported in [13]. Object model instances are instance of only one class, and must conform exactly to the structure (properties and methods) of their class definition, whereas in RDF/OWL instances may easily deviate from their class definitions or have no such definitions at all. The purpose of our object ontology mapping approach is not to provide a solution to the impedance mismatch between object and RDF. We

rather consider ontologies as schema for online RDF data sources that help us to identify required data to be used by an application. The mapping language we have developed helps defining rules for instantiating objects from RDF data, and rules for publishing objects as RDF data. Figure 1 presents the domain object model that will be used as an example throughout this article, as well as the ontology associated with it.

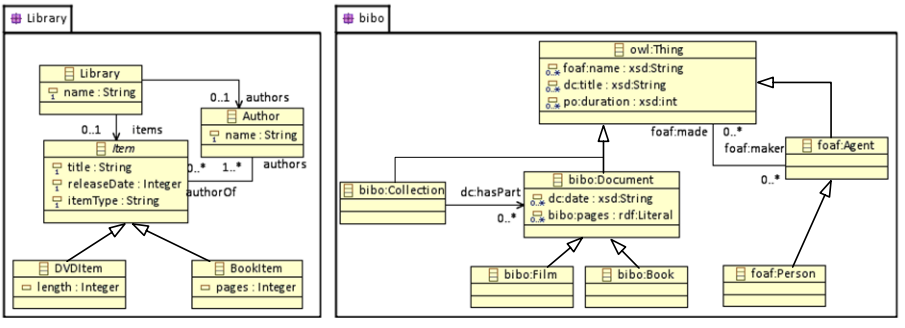


Fig. 1. Domain object model (left) and an excerpt of the bibo1 ontology (right)

3.2 Object Ontology Mapping Language

We propose MEO (Mapping Ecore to OWL), a declarative language for the definition of binary mappings between domain object models and ontologies. This mapping is based on the well-defined semantics of the ATL language. The mapping language is compiled into two ATL transformations, *via* a couple of higher order transformations (refer to [10] for more details). The first generated transformation takes the object model as input, and produces an RDF model (according to the ontology vocabulary). The second generated transformation deals with the opposite way. It takes an RDF model as input and produces a model that conforms to the application domain model, thus finally enabling the retrieval of objects from RDF data sources.

Our mapping language has its own abstract and concrete syntaxes developed by using the TCS toolkit [15]. An excerpt of a mapping is given in Listing 1. The mapping is composed of two mapping rules indicating the correspondences between on the one hand the object model class *Author* and the ontology class *foaf:Person*, and on the other hand, the object model class *DVDItem* with *bibo:Film*. Let's note that two ontologies are used here, since the *bibo* ontology uses terms from the *foaf*[2] ontology.

A mapping rule specifies variables to identify the classes to be mapped. Each variable can be initialized thanks to OCL expressions. The *classMap* operation indicates which classes are mapped, while the *propertyMap* operation maps properties. The *get()* and *put()* operations are others kind of *propertyMap()* operations to be used when complex property mappings are to be defined.

[1] http://bibotools.googlecode.com/svn/bibo-ontology/trunk/doc/index.html
[2] http://xmlns.com/foaf/spec/

```
1   mapping library
2   models = {lib : 'http://org.example.library'}
3   ontology = {
4           bibo: 'http://purl.org/ontology/bibo/,
5           foaf: 'http://xmlns.org/foaf/0.1/'
6   }
9   rule Author2Person {
10        def a is lib:Author
11        def b is foaf:Person
12        def getName is String as b.foaf:name.firstLiteral()
13        def putName is String as a.name
14        classMap(a, b)
15        put(putName, b.foaf:name)
16        get(a.name, getName)
17        propertyMap(a.authorOf, b.foaf:made)
18  }
19  rule DVDItem2Film {
20        def a is lib:DVDItem
21        def b is bibo:Film
22        classMap(a, b)
23        propertyMap(a.title, b.dc:title)
24        propertyMap(a.length, b.po:duration)
25        propertyMap(a.authors, b.foaf:maker)
    }
```

Listing 1. Excerpt of a mapping declaration in MEO language

3.3 Specification, Implementation and Execution

The object ontology mapping solution provides a Java library offering common methods for loading/saving objects to/from RDF data sources, as well as a query engine enabling the query of RDF data source through an object oriented query language. This library makes use of models and model transformations at runtime. Using this solution spans three levels:

Specification. The specification step comprises the definition of the object model of the application. This object model is the data model, and is defined thanks to the Eclipse Modeling Framework (EMF) in Ecore. The second specification step is the selection of the ontology that will be mapped to the object model. This ontology can either be a custom one, designed as a simple mirror of the object model, or a common previously existing one, possibly extended by extra concepts. The third specification step is the definition of the mapping between the object model and the ontology thanks to the MEO language. The mapping can be trivial if the object model and the ontology are close to one another, or more complex in case there exist significant differences.

Implementation. The implementation step uses the Java library offered by the object/ontology mapping solution for the Java application which has to access RDF data sources. This library provides methods to save/load objects to/from RDF data sources. In this implementation step, the preferred scenario supposes the generation of POJO (Plain Old Java Object) classes from the Ecore model defined during the specification step.

Execution. The object ontology mapping solution is performed during the execution of the Java application (during loading/saving objects). The execution step uses the ATL engine at runtime. ATL model transformations are generated from our mapping language and executed when translation of objects to or from RDF is required. The ATL engine is also used at runtime for the query rewriting process as presented in this paper.

4 Query Languages

This section presents the query languages used by our approach: an object-oriented language based on OQL and a query language for RDF data (SPARQL). The object-oriented language chosen for this implementation is HQL (Hibernate Query Language) [2].

4.1 Object Query Language

The OQL language is a standard developed by the ODMG (Object Data Management Group) for the expression of queries on object oriented databases. It suffers from too much complexity and has never been fully implemented. Several variants of this standard exist in implementations such as: JDOQL, EJBQL or HQL.

Here, we partially redefine the implementation of the HQL language with the help of MDE toolkits. A metamodel is developed from the grammar of the HQL language. The textual syntax of the language is defined, according to the metamodel, using the TCS toolkit. The execution of a query is made either with the Hibernate framework [2] when needing to retrieve objects form relational data, or by using model transformations joint with our object ontology mapping tool when needing to retrieve objects from RDF data. The latter solution is the one presented in this article.

We limit our presentation to HQL queries of type *SELECT*. The following figure shows the representation of a query of such a type in conformance to the HQL metamodel. A *SELECT* statement is composed of four clauses: *SelectFromClause*, *WhereClause*, *OrderByClause* and *GroupByClause*.

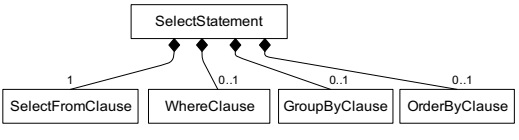


Fig. 2. The Select Statement representation in the HQL metamodel

In a *SELECT* query, only the *SelectFromClause* element is mandatory. Examples of valid HQL queries (according to our HQL metamodel) are given below:

```
(a) from Library
(b) select a.name from Author a, a.authorOf item
       where item.publishDate > '2000'
```

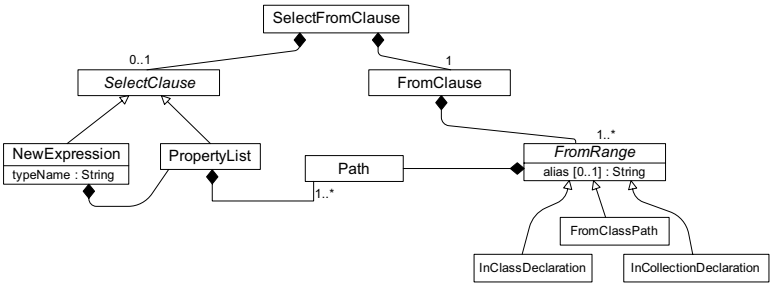


Fig. 3. Representation of the Select From Clause in the HQL metamodel

HQL query results are either lists of objects or lists of values. Query (a) returns the list of objects of type `Library`. Query (b) returns a list of values from the property *`name`* of all objects of type `Author` having published an item after `2000`.

The *SelectFromClause* representation is given in Figure 3. It is composed of an optional *SelectClause* and a mandatory *FromClause*. A *SelectClause* may be of two types: *NewExpression* is a clause which creates a new object. *PropertyList* contains the list of values or objects resulting from the query. Values are path expressions composed of one or more terms. A term is either an alias for a type of the object model, or a property of the object model. A path permits to browse the object model.

The *FromClause* allows selecting these elements from the object model that will be used in the query. The declaration of these elements is achieved either by indicating the desired class in the object model (represented in the HQL metamodel by *FromClassPath*) or indicating an association. Each element of a *FromClause* is identified by an alias that will be used for specifying paths in the query.

The *WhereClause* element represents the constraint part of the query. A *WhereClause* basically is an expression which can either be a binary expression (and, or), an operator expression (=, <, =<, >, >=), a path expression, or a value (string, integer, boolean).

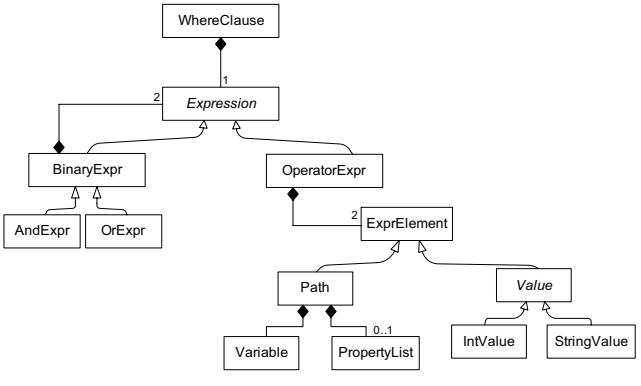


Fig. 4. Representation of the Where Clause in the HQL metamodel

4.2 RDF Query Language (SPARQL)

SPARQL enables the retrieval of data available on the Web in the form of RDF triples. RDF models data as a directed labeled graph that represents the information on the Web. SPARQL can be used to express queries across various RDF data sources, potentially native RDF data, or RDF data generated automatically, e.g. *via* a SPARQL endpoint. A SPARQL endpoint is a server managing the conversion of data (relational or other) into RDF, when answering the receipt of a SPARQL query.

SPARQL is based on the concept of graph patterns for the selection of RDF triples. A pattern is a triple composed of one or more variables. An example of SPARQL query is given below.

```
(c)  select ?o where { ?s rdf:type ?o }
(d)  construct {
            ?s dc:title ?t ;
               po:duration ?n ;
               bibo:pages ?p .
     }
     where {
            ?s dc:title ?t .
            ?s dc:date ?d .
            filter (?d > '2000') .
            optional {?s po:duration ?n}
            optional {?s bibo:pages ?p}
     }
```

A SPARQL query returns a list of values (*ResultSet*) or an RDF graph. For example, Query (c), of type SELECT, returns a list of values corresponding to the identifier (URI) of the types of all resources in an RDF graph. Query (d) returns an RDF graph constituted by a set of patterns built in accordance to the set of patterns defined in the WHERE clause. Let's note that the WHERE clause in Query (d), includes a test value using a filter, and two optional patterns. An optional pattern is used to select a pattern potentially not present in the graph.

To carry our approach out, we defined the SPARQL metamodel from the SPARQL grammar. We derived the concrete syntax from the metamodel and used the TCS toolkit.

Figure 5 shows the SPARQL metamodel part and addresses the different possible types for a query. In this study, we are only interested in queries of types SELECT and CONSTRUCT.

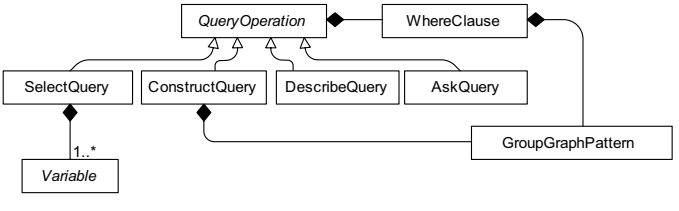


Fig. 5. The different types of Query Operation in the SPARQL metamodel

SPARQL basic concept is the triple. A set of triples forms a graph pattern. The representation of this concept in the SPARQL metamodel is given in Figure 6. The *GroupGraphPattern* consists of a set of graph patterns (*GraphPattern*) representing the various kinds of graph pattern.

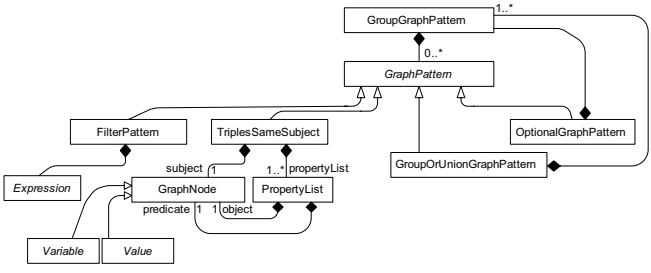


Fig. 6. Representation of the Group Graph Pattern in the SPARQL metamodel

The simpler one is the triple represented by the element *TriplesSameSubject* which includes a subject and one or more associated properties. Other types are *GroupOrUnionGraphPattern* for union of patterns; *OptionalGraphPattern* for optional patterns and *FilterPattern* for specifying filters. Graph patterns are created from nodes. A node is either a variable (named or free) or a primitive type. A named variable is identified by an URI.

5 Model Transformations

This section presents the implementation of our solution for rewriting HQL queries into SPARQL and *vice versa*. We use the model transformation language ATL.

5.1 Rewriting HQL in SPARQL

Rewriting into SPARQL an HQL query expressed in the terms of an object model is carried out by a model transformation that needs two inputs: the mapping defined between the object model and the ontology and the HQL query itself. The following figure depicts the overall rewriting process.

The HQL query is first translated into a model that conforms to the HQL metamodel. This model is used as an input by *HQL2SPARQL.atl*. The output is the corresponding SPARQL model. During the transformation execution, the object ontology mapping is used to identify the correspondences between object model terms and ontology terms. The SPARQL model is sent to a relevant RDF data source. The execution of the query returns an RDF graph. This graph is then transformed into a model conforming to the RDF metamodel. The RDF model is used as input by rdf2model.atl. This transformation is generated by a high order transformation from the object ontology mapping. The execution of *rdf2model.atl* supplies the resulting object graph.

The HQL query returns a list of items or a list of values or a list of values and objects. Hereafter, we detail the first case, and only outline the second. The last case is not yet supported by our implementation. Figure 7 depicts the case where the SPARQL query returns an RDF graph.

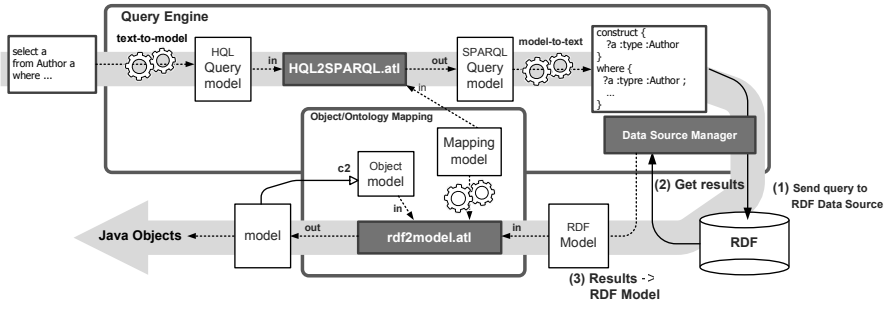


Fig. 7. Query execution process : From HQL to SPARQL

Case 1. The query returns a list of values. In this case the HQL query must be translated into a SPARQL query of type SELECT. This type of query returns a list of values and not an RDF graph. The result of the HQL query is the result of the SPARQL query having been generated. No transformation is needed since the data retrieved actually are plain values.

Case 2. The query returns a list of objects. In this case the HQL query must be translated into a SPARQL query of type CONSTRUCT. This type of SPARQL query returns an RDF graph. The latter will be processed by our object ontology mapping engine, so as to transform the RDF graph into an object model.

The implementation of all cases above is driven by two distinct model transformations. The transformation rules presented below refer to Case 2. However most of the rules are common to both cases, particularly for what concerns the generation of graph patterns from HQL path expressions. Let's explain more about the transformation rules, and let's consider the following query which is defined on the object model presented in Fig. 1 at Section 3.1.

```
(e) select item, author from Item item, item.authors author
      where item.releaseDate > '2000'
```

Let *Q* be the HQL query, *Q'* the resulting SPARQL query and *M* the object ontology mapping. The transformation HQL2SPARQL is hence defined by:

$$SPARQL2HQL(Q : HQL, M : MEO) \rightarrow Q' : SPARQL$$

Rule A: For each *Path* identifying an object belonging to a *PropertyList* element in the *SelectClause* of query *Q (if any)*, a set of triple patterns (*TriplesSameSubject*) is generated and added to a *ConstructQuery* element in query *Q'*.

```
select item, author from
    BookItem item ...
=>
construct {
    ?item rdf:type bibo:Book ;
          dc:title ?itTitle ;
          dc:date ?itDate .
    ?item foaf:maker ?author .
    ?author rdf:type :Author ;
      ...
}
```

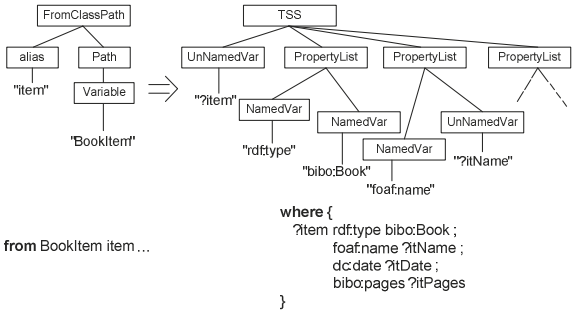
The set of triples enables to retrieve each property belonging to a given RDF property. The set comprises a triple identifying the type of the resource. This type is retrieved according to the mapping. Other triple patterns correspond to the properties required by the mapping.

Rule B: For each *FromClassPath* element from Query *Q*, having a *Path* size equal to 1 and identifying an object, a set of triple patterns (*TriplesSameSubject*) is generated and added to the *WhereClause* in Query *Q'* (cf. figure below).

Rule C: For each *FromClassPath* from Query *Q* having a *Path* element of size *N (N>1)* and being a valid expression path according to the object model, a set of *N-1* triple patterns is created in Query *Q'* by applying recursively the previous rule. (example: `a.b.c => ?a :b ?ab . ?ab :c ?bc`).

Rule D: For each expression in the *WhereClause* of Query *Q* being of type *OperatorExpr and* having the symbol '=' as operator, a corresponding triple pattern is created in the *WhereClause* of Query *Q'*. (example: `s.p = o => ?s :p ?o`)

Rule E: For each expression of type *OperatorExpr* in the *WhereClause* of Query *Q and* having an operator that belongs to the following list: (> | >= | < | =<), a triple pattern and a filter pattern are created in the *WhereClause* of Query *Q'*. The filter pattern contains the expression occurring in the *OperatorExpr*.
(example: `s.p > val => ?s :p ?sp . filter(?sp > val)`)



5.2 Rewriting SPARQL in HQL

Information systems persistency is usually achieved through relational database, XML files, etc… However, using an RDF representation is a way to facilitate data integration and matching, as promoted by the Semantic Web architecture. Semantic Web technologies provide and enable representation of existing data via a common

vocabulary (ontologies) that can be extended so that an additional vocabulary could be taken into. Data represented via RDF graphs can be interlinked with each other, allowing an easy navigation within a graph representing the global data on the Web.

Providing tools allowing an efficient and transparent transformation of existing data into RDF is a most important issue Through the abstraction offered by the object model, we can take advantage of mappings, transformations, converters, etc., that already exists between the object abstraction and data sources such as relational, XML, etc. By adding an object ontology mapping tool to these solutions, we can get a complete conversion chain between heterogeneous data sources and Semantic Web data. Query rewriting between SPARQL and an object query language (such as HQL) allows an on the fly generation of RDF data and thus makes it possible to keep data in existing databases and avoid data replication with its entailed lack of synchronization and integrity.

The rewriting process from SPARQL to HQL supposes two prerequisites. First an object model must have been explicitly or implicitly identified (e.g. application classes in the latter case). Second, an ontology is associated to this object model (see example Figure 1). Finally, a SPARQL endpoint has been implemented in order to connect our application to the Web, and thus enable receipt and processing of SPARQL queries.

The SPARQL endpoint is a Web server receiving queries from others Web applications. SPARQL queries received are injected in the form of SPARQL model thanks to a text to model transformation, as depict in figure 8. This SPARQL model is processed by a set of model transformations and result in a HQL model. During those transformations, the object ontology mapping helps to determine how terms used in the SPARQL queries (ontology terms) are converted in terms for the HQL queries (object mode terms). The resulting HQL query is used by a Java application, and may be executed thanks to the Hibernate framework on a relational database. Resulting objects are translated in RDF thanks to our object ontology mapping solution, as depicted by figure 8.

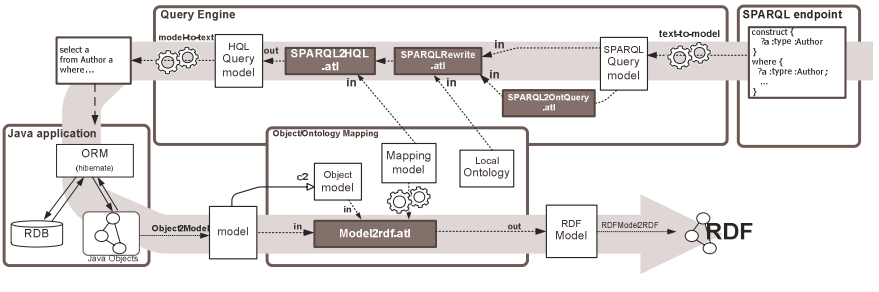


Fig. 8. Query execution process: From SPARQL to HQL

The rewriting process of a SPARQL query into the object formalism chains three steps, corresponding to three model transformations. Let's illustrate our transformation rules, by treating the following SPARQL query as an example:

```
(f)  construct {?a ?p1 ?o1 .?c ?p2 ?o2 }
     where {
       ?a dc:date ?c . filter(?c > '2000') .
        ?a foaf:maker ?b
     }
```

5.2.1 Step 1: Identifying Types in Initial Query

The first step is to identify the types of each variable in the query. All potential types are determined by querying the ontology associated with the object model. For example, Query (f) uses three unbounded variables: ?a, ?b and ?c. By parsing the *WhereClause* of Query (f), we notice that there is no triple pattern using the RDF property *rdf:type* which would explicitly indicate the type of variables. However, the triple patterns use named variables as predicates (named variables are URIs). So, we can infer the types by identifying the domain and range of the triple pattern predicates. This can be done by simply querying the ontology. To do so, we rewrite the query from the initial one, by adding the unknown types in the select clause. Performing the new query will retrieve the missing information about types. More precisely, the rewriting process is handled by the model transformation *SPARQL2OntQuery.atl* (see Figure 8). For example query (f) is rewritten as follows:

```
(g) select
    ?dateDom, ?dateRang, ?makerDom, ?makerRang
    where {
       dc:date rdfs:domain ?dateDom ;
               rdfs:range ?dateRang .
        foaf:maker rdfs:domain ?makerDom ;
                  rdfs:range ?makerRang .
   }
```

The *SPARQL2OntQuery* transformation takes the initial query *Q* as input and is defined as follows: *SPARQL2OntQuery(Q : SPARQL)* → *Qt : SPARQL*

Running Query (g) over the ontology provides all valid types for the free variables. The result is serialized by a SPARQL Engine implementation (Jena ARQ[3]) in an XML format (SPARQL Results XML Format[4]). A metamodel of this format, marked as SRF, has been defined in order to allow the next step transformation to reuse the former results.

5.2.2 Step 2: Refining the Initial SPARQL Query by Adding Types

The second step takes the previous results into account. Types that have been inferred by performing *SPARQL2OntQuery* are inserted into the initial SPARQL query so as to bind the free variables to their valid types. This is done by the model transformation *SPARQLRewrite.atl* (see figure 8). This transformation takes the initial query *Q*, the ontology *O*, and the result *Rt* produces by query *Qrw* as inputs.

SPARQLRewrite(Q :SPARQL, O :OWL, Rt :SRF) → *Qrw : SPARQL*

[3] http://jena.sourceforge.net/ARQ/
[4] http://www.w3.org/TR/rdf-sparql-XMLres/

The query *Qrw* is identical to the initial query *Q* except that triple graph patterns identifying variables types are added. This set of additional triples patterns is denoted *Tp*. For each variable v in *Q*, having Type *t* according to *Rt*, there exists a pattern *p* such that p = `{?s rdf:type t}`. Thus the query (f) rewrites as:

```
(h) construct {?a ?p1 ?o1 . ?c ?p2 ?o2 }
    where { ?a dc:date ?c . filter(?c > '2000') .
            ?a foaf:maker ?b ;
            ?a rdf:type bibo:Film .
            ?a rdf:type bibo:Book .
            ?b rdf:type foaf:Person
    }
```

5.2.3 Step 3: Transformation Rules for SPARQL to HQL

The last step encompasses the transformation of SPARQL into HQL. This operation is trivial once the types of the variables are known i.e. when Step 2 is completed. The rewriting process is performed by the model transformation *SPARQL2HQL.atl*. This transformation takes the query *Qrw* and the object ontology mapping *M* as inputs.

$$SPARQL2HQL(Qrw : SPARQL, M : MEO) \rightarrow Q' : HQL$$

The transformation rules address the translation of SPARQL triples graph patterns into HQL path expressions. The main transformation rules are the following:

Rule A: For each triple pattern `{?s rdf:type <URI>}` belonging to the *WhereClause* in query *Qrw*, a *FromClassPath* (`from ClassName ?s`) is created where `ClassName <- map(<URI>)` and *map* the mapping function from ontology to object model.

Rule B: For each triple pattern `{?s ?p ?o}` belonging to the *WhereClause* in Query *Qrw* and having as predicate an *ObjectProperty* (property with an RDF resource as range), the triple is translated into an *OperatorExp* `(s.p = o)` in query *Q'*. The expression in *OperatorExp* is the *Path* formed by the subject and predicate of the corresponding triple pattern and has for RHS expression its object.

Rule C: For each triple pattern `{?s ?p val}` belonging to the *WhereClause* in Query *Qrw* and *val* an RDF::Literal then the triple is translated into an *OperatorExp* `(s.p = val)` in query *Q'*.

Rule D: For each triple pattern `{?s ?p ?o . filter(?o op val)}` belonging to the *WhereClause* in Query *Qrw* then an *OperatorExp* `(s.p op val)` is created in query *Q'*.

6 Conclusion

In this paper, we presented a query rewriting process implemented by model transformations. These transformations exploit a mapping model that describes the relationship

between the elements of a model object with those of an ontology. It was also necessary to define the metamodels of the two query languages handled in these transformations: HQL as an object oriented query language and SPARQL as a graph pattern query language for the RDF data model.

The Model Driven Engineering makes it possible to manage complexity inherent in the translation of requests built on quite different data models. The rules of transformation presented in this paper, implemented using ATL language, include transformations from HQL to SPARQL and opposite directions. The main goal of this work is to facilitate the use and the enrichment of the many collections of RDF data available on the Web without having simultaneously to master the object technologies and Semantic Web technologies.

This work is fully implemented, but it has not been heavily evaluated and so a comparison with others similar approaches, in terms of response time and scalability has not yet been done. Future works include the evaluation of the tool and its extension so as to cope with transformation rules taking more complex query languages features into account, namely *join* for HQL and 'optional' and 'union' patterns for SPARQL.

References

1. Ankolekar, A., Krötzsch, M., Tran, T., Vrandecic, D.: The two cultures: Mashing up Web 2.0 and the Semantic Web. Web Semantics: Science, Services and Agents on the World Wide Web 6, 70–75 (2008)
2. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications (2006)
3. Bechhofer, S., Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference. W3C Recommendation 10, 2006–10 (2004)
4. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284, 28–37 (2001)
5. Bizer, C., Seaborne, A.: D2RQ: treating non-RDF databases as virtual RDF graphs. In: International Semantic Web Conference ISWC (posters) (2004)
6. Bizer, C., Heath, T., Ayers, D., Raimond, Y.: Interlinking Open Data on the Web Demonstrations Track. In: 4th European Semantic Web Conference, Innsbruck, Austria (2007)
7. Corno, W., Corcoglioniti, F., Celino, I., Della Valle, E.: Exposing Heterogeneous Data Sources as SPARQL Endpoints through an Object-Oriented Abstraction. In: Asian Semantic Web Conference (ASWC 2008), pp. 434–448 (2008)
8. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Proceedings of the 1st Conference on Social Semantic Web. GI-Edition- Lecture Notes in Informatics (LNI), vol. P-113. Bonner Kollen Verlag (2007) ISSN 1617-5468
9. Ghawi, R., Cullot, N.: Database-to-ontology mapping generation for semantic interoperability, 2007. In: Third International Workshop on Database Interoperability, InterDB (2007)
10. Hillairet, G., Bertrand, F., Lafaye, J.Y.: MDE for publishing Data on the Semantic Web, Transform and Weaving Ontologies in MDE (TWOMDE). In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)

11. de Laborda, C.P., Conrad, S.: Bringing Relational Data into the SemanticWeb using SPARQL and Relational. OWL. IEEE Computer Society, Washington (2006)
12. Lassila, O., Swick, R.R.: Resource Description Framework (RDF) Model and Syntax Specification (1999)
13. Oren, E., Heitmann, B., Decker, S.: ActiveRDF: Embedding Semantic Web data into object-oriented languages. In: Web Semantics: Science, Services and Agents on the World Wide Web (2008)
14. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
15. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 249–254 (2006)
16. Prud'hommeaux, E., Seaborne, A.: others: SPARQL Query Language for RDF. W3C Recommendation (2008)
17. Prud'hommeaux, E.: Adding SPARQL Support to MySQL (2006)
18. Rodriguez, J.B., Corcho, O., Gomez-Perez, A.: R2o: an extensible and semantically based database-to-ontology mapping language. In: SWDB (2004)
19. Rodriguez, J.B., Gomez-Perez, A.: Upgrading relational legacy data to the semantic web. In: Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M. (eds.) WWW, pp. 1069–1070. ACM, New York (2006)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)