Exposing heterogeneous data sources as SPARQL endpoints through an object-oriented abstraction

Walter Corno, Francesco Corcoglioniti, Irene Celino, and Emanuele Della Valle

CEFRIEL - Politecnico di Milano,
Via Fucini 2, 20133 Milano, Italy
email: walter.corno@students.cefriel.it,
for other authors {name.surname}@cefriel.it
website: http://swa.cefriel.it

Abstract. The Web of Data vision raises the problem of how to expose existing data sources on the Web without requiring heavy manual work. In this paper, we present our approach to facilitate SPARQL queries over heterogeneous data sources.
We propose the use of an object-oriented abstraction which can be automatically mapped and translated into an ontological one; this approach, on the one hand, helps data managers to disclose their sources without the need of a deep understanding of Semantic Web technologies and standards and, on the other hand, takes advantage of object-relational mapping (ORM) technologies and tools to deal with different types of data sources (relational DBs, but also XML sources, object-oriented DBs, LDAP, etc.).
We introduce both the theoretical foundations of our solution, with the analysis of the relation and mapping between SPARQL algebra and *monoid comprehension calculus* (the formalism behind object queries), and the implementation we are using to prove the feasibility and the benefits of our approach and to compare it with alternative methods.

1 Introduction

The Semantic Web has as ultimate goal the construction of a Web of Data, i.e. a Web of interlinked information expressed and published in a machine-readable format which enables automatic processing and advanced manipulation of the data itself. In this scenario, initiatives like the Linking Open Data community project[1], guidelines and tutorials on how to publish data on the (Semantic) Web [1, 2] and standards for querying this Web of Data like SPARQL [3, 4] play a central role in the realization of the Semantic Web vision.

To achieve this aim, however, it is necessary to find an easy and automated – as much as possible – way to expose existing data sources on the Web. To this

[1] http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

end, two big classes of approaches are being studied to help data managers to prepare their data sources for the Semantic Web: conversion and wrapping.

With *conversion* we mean all the techniques to effect a complete translation of the data source from its native format to a Semantic Web model (pure RDF or RDF triples described by some kind of ontologies in RDFS/OWL). This approach assures a complete replication and porting of the data but raises several concerns like frequency of re-conversion, synchronization, conversion processing and space occupation.

With *wrapping*, on the other hand, we mean all the techniques aimed at building an abstraction layer over the original data source which hides the underlying format and structure and exposes a SPARQL endpoint to be queried. This approach requires the run-time translation of the SPARQL query request to the source's specific query language and the run-time conversion of the query results back to the requester. Several proposals, that are gaining ground within the Semantic Web community, belong to this wrapping approach through the declaration of a mapping between the original data format and the respective RDF data model. Most of those approaches, however, deal with the wrapping of relational databases (see Section 2) and do not consider other kinds of data sources. Moreover, this direct mapping requires the developer to have quite a deep understanding of Semantic Web languages, technologies and formats to express the declaration of correspondences.

In this paper, we present our proposal for a new approach that tries to overcome the aforementioned problems. Our approach belongs to the wrapping category, but tries to give a solution to the heterogeneity of data sources as well as to the problem of adoption by the larger community of developers. For the former issue, we provide a solution based on the availability of approaches and tools to wrap different data sources with an object-oriented abstraction; for the latter problem, we propose an automatic mapping between the object-oriented model (in ODL) and the correspondent one at the ontological level (in OWL-DL, see Section 3). Our approach is both theoretically sound – because of the affinity between object-orientation and ontology modelling and because of the accordance of the respective formalisms (SPARQL algebra [3] and monoid comprehension calculus [5], see also Section 2) – and technologically and technically practicable – because we realized SPOON, the reference implementation of our approach.

The remainder of the paper is structured as follows: Section 2 introduces related approaches and theoretical foundations; Section 3 explains the automatic mapping between object-oriented models and ontologies (with its constraints); we illustrate our approach to query translation in Section 4 and our implementation and evaluation in Section 5; concluding remarks and next steps are finally presented in Section 6.

2 Related work

In order to enable a faster expansion of the Web of Data, in the last few years some efforts arose with the aim to expose existing datasources, especially relational databases (RDBMs), on the Web and to query them through SPARQL [3].

For instance, Cyganiak and Bizer studied similarities between SPARQL algebra and relational algebra [6] and developed D2RQ [7] and D2R Server, to build SPARQL endpoints over RDBMs. SquirrelRDF[2] exposes both relational and LDAP sources, but it is still incomplete; SPASQL [8] is a MySQL module that adds native SPARQL support to the database; Relational.OWL [9], Virtuoso Universal Server [10], R2O [11] and DB2OWL [12] are other projects that aim to expose relational data on the Web.

While relational databases are the most widespread, the most common programming paradigm is object-oriented (OO) programming. Since the OODBMS Manifesto [13] many projects developed proprietary object datasources (e.g. O2, Versant, EyeDB, and so on[3]), but none of them strictly follows the ODMG Standard [14], so these technologies failed in being either widely used or interoperable. As a consequence, new technologies were born from the cited ones: the Object-Relational Mappings (ORMs), that allow to use relational sources as if they were object datasources and to query them in an object-oriented way. Well-known ORMs are Hibernate, JPOX, iBatis SQL Maps and Kodo[4]. Among these, JPOX and Kodo implement the JDO specification [15], a standard Java-based model of persistence, that allows to use not only RDBMs but also many other types of source (OODBMS, XML, flat files and so on)[5]. Even if different in syntax and characteristics, all the object query languages developed so far are based on the Object Query Language (OQL) developed by the ODMG consortium [14]. The *monoid comprehension calculus* [5] is a framework for query processing and optimization supporting the full expressiveness of object queries; it can be considered as a common formalism and theoretical foundation for all OQL-like languages. This formalism has been used in [16] to translate queries in description logics to object-oriented ones.

3 Schema and data mapping

The first step of our proposed approach is to help the data manager to expose his data-source schema (already wrapped by an ORM) as an ontological model. To this end, we propose to adopt a specific mapping strategy to make this step completely automatic (albeit some restrictions/constraints on the OO model).

Object-oriented model is much more similar to ontological model than relational one. In particular, these models share a common set of primitives (e.g. *classes*, *properties*, *inheritance*,...), and can describe relationships between classes directly, whereas the relational model may require complex expedients such as the use of *join tables*.

OO and ontological models are not fully equivalent, as shown in [17, 18] (e.g. single vs. multiple inheritance and local vs. global properties); however the

[2] `http://jena.sourceforge.net/SquirrelRDF/`

[3] Versant Object DB: `http://www.versant.com/`, EyeDB `http://www.eyedb.org/`

[4] Hibernate: `http://www.hibernate.org/`, JPOX: `http://www.jpox.org/`, Apache iBatis: `http://ibatis.apache.org/`, BEA Kodo: `http://bea.com/kodo/`

[5] For these reasons in our implementation we chose JPOX as ORM tool.

issues highlighted in these works are only relevant for the problem of describing an existing ontology as an object model (due to some limitations of the OO model), while in our approach we deal with the opposite problem (i.e., to expose an OO model as an ontology).

In our approach we propose a one-to-one mapping as simple as possible (similar to the one shown in [19]), because we aim to automatize it, simplifying the development process. We use ODL [14] as OO formalism and a subset of OWL-DL [20] (represented by the constructs in Table 1 and disjointness, as explained below) as the ontology language. The schema mapping is described in Table 1.

Concept	ODL	OWL-DL
Class	class	owl:Class
Subclass	class A extends B	rdfs:subClassOf
Property	attribute/relationship	owl:DatatypeProperty/ObjectProperty
Inverse relationship	inverse	owl:inverseOf
Property domain	*implicit*	rdfs:domain
Property range	*property type*	rdfs:range
Primitive types	int, double,...	*XSD datatypes*
Functional property	*non-collection types*	owl:FunctionalProperty
Non-functional prop.	set<T>[6]	*implicit*

Table 1. Schema mapping

In addition to these correspondences, we add *disjoint* constraints to the ontology because objects can belong only to a single OO class (with its parents):

$$\forall \; class\ C_1, C_2 : \nexists \; class\ C \;\; subClassOf\, C_1, C_2, \\ generate\ \langle C_1 \text{ owl:disjointWith } C_2 \rangle$$

Moving from schema to instance mapping, primitive instances are mapped to RDF literals, while to map objects we need a way to create URIs for them (because they become RDF resources). The simplest approach we adopt is to combine a fixed *namespace* with a variable *local name*, formed by the values of a particular *ID* property; we prefer not to use the OIDs commonly employed in OODBMS, due to their limited support among ORMs. Object attributes and relationships are then translated into RDF triples, using the corresponding predicates as defined by the schema mapping.

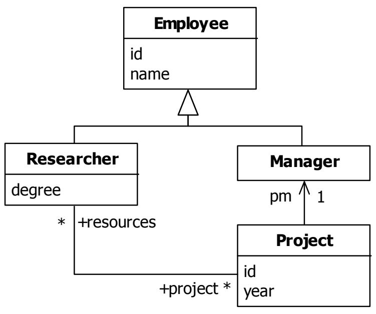
To keep the mapping simple and ease its automatization, we introduce some constraints on the OO model:

– all classes have to contain an alphanumeric property *ID*, with a unique value (in class hierarchies it can be inherited from a parent class).
– OO properties having the same name in unrelated classes can only be mapped to different ontological predicates, thus having distinct semantics.
– *collection* properties are limited to the *set* type (no *bag*, *list* or *map*).

[6] *Set* is the collection type and T is the type of the elements contained in the collection.

– interfaces are not supported (and thus multiple inheritance).
– all classes have an *extent* in order to be directly used in OO queries (see [14] for *extent* definition).

Figure 1 shows the translation to an OWL ontology of a simple ODL schema, which will be used as a running example throughout the paper.



```
class Employee (extent Employees) {
    attribute string id;
    attribute string name;
}
class Researcher extends Employee
                (extent Researchers) {
    attribute string degree;
    relationship set<Project> project
                inverse Project:resources;
}
class Manager extends Employee
            (extent Managers) {
}
class Project (extent Projects) {
    attribute string id;
    attribute integer year;
    attribute Manager pm;
    relationship set<Researcher> resources
                inverse Researcher::project;
}
```

(a)

```
:Project a owl:Class .
:Employee a owl:Class .
:Researcher a owl:Class ;
    rdfs:subClassOf :Employee .
:Manager a owl:Class ;
    rdfs:subClassOf :Employee .
:hasName a owl:DatatypeProperty ;
    a owl:FunctionalProperty ;
    rdfs:domain Employee ;
    rdfs:range xsd:string .
:hasDegree a owl:DatatypeProperty;
    a owl:FunctionalProperty ;
    rdfs:domain :Researcher ;
    rdfs:range xsd:string .
:hasYear a owl:DatatypeProperty ;
    a owl:FunctionalProperty ;
    rdfs:domain Project ;
    rdfs:range xsd:integer.
:hasProject a owl:ObjectProperty ;
    rdfs:domain :Researcher ;
    rdfs:range :Project ;
    owl:inverseOf hasResource .
:hasResource a owl:ObjectProperty ;
    rdfs:domain :Project ;
    rdfs:range :Researcher ;
    owl:inverseOf :hasProject .
:hasPM a owl:ObjectProperty ;
    a owl:FunctionalProperty ;
    rdfs:domain :Project ;
    rdfs:range :Manager .

:hasID a owl:DatatypeProperty ;
    a owl:FunctionalProperty ;
    rdfs:range xsd:string .
```

(b)

Fig. 1. Running example schema mapped to the corresponding ontology.

4 Query translation

In this section we present our framework to translate a SPARQL query into one or a few object queries. The general process is shown in Figure 2, sketched hereafter and explained in details throughout the whole section.

When a new SPARQL query is sent to our system, first we perform an *analysis* process both at the *syntactic* and *semantic* levels. During *syntactic analysis*

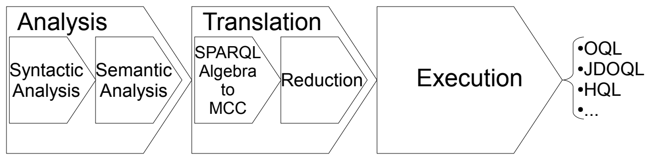


Fig. 2. The general query processing framework

the query is parsed, checked for syntactic errors and then translated into its equivalent SPARQL algebraic form [3], which is then *normalized*. In this format, a query is represented as a tree composed of *basic graph patterns* (BGP) as leaves and of the algebraic operators *filter* (σ_{pred}), *union* ($\cup$), *join* ($\bowtie$), *left join* ($⟕_{pred}$) and *diff* ($-_{pred}$)[7] as internal nodes. Then we perform *semantic analysis*: we apply some checks and rewriting rules to ensure that the query can be processed by the next phases. Variables on predicates are resolved and variables on subjects/objects are assigned to the corresponding OO classes.

The second step is the *core* one of our framework: the *query translation*. In this step the SPARQL algebraic form of the query is translated in a *monoid comprehension calculus* [5] expression, so the initial SPARQL query is now expressed as a query on the OO model. The translation starts processing basic graph patterns (BGPs) and then translating each SPARQL algebraic operator we meet when traversing the SPARQL algebraic form of the query in a *bottom-up* approach. When this translation is completed, we apply the *normalization rules* demonstrated in [5] to the global expression (*reduction* phase), so that we get a simpler expression (as we will see, a *union* of *monoid comprehensions*).

The last step is the *query execution*. In this step the obtained *union* of *monoid comprehensions* is translated into queries of the particular OO query language used for the implementation of the framework and then executed. Eventually the final result-set is translated into the one compatible with the original SPARQL query (*select*, *construct*, *describe*, *ask*).

In the remaining of this section we explain these three steps in detail, continuing the running example introduced in Section 3 with the following SPARQL query, whose effect is to return the URI, the names and (optionally) the degree of all the employee related to projects of 2006 and later.

```
SELECT ?e ?n ?d
WHERE {
   ?p hasYear ?y ;
      ?r ?e .
   ?e hasName ?n .
   OPTIONAL { ?e hasDegree ?d }
   FILTER ( ?y >= 2006 )
}
```

[7] To ease the notation, we borrow the symbols of *relational algebra.*

4.1 Analysis

The analysis phase takes care of parsing, checking and transforming the SPARQL query in order to prepare it for the subsequent translation phase. Query analysis is performed both at the *syntactic* and *semantic* levels.

Syntactic analysis. The first step is to parse the input query string, check its syntax and produce as output its equivalent representation in SPARQL algebra [3], as shown in Figure 4 (a) for the query of the running example. The parsed algebraic representation is then normalized, in order to "collapse" as far as possible the BGPs of the query and to reach a form easier to analyse and translate. The *normalization procedure* consists of three steps:

1. *Left joins replacement*, with a combination of *union*, *join*, *filter* and *diff* operations, according to the rule [3]:

$$A \ltimes_{pred} B \Rightarrow \sigma_{pred}(A \bowtie B) \cup (A -_{pred} B) \tag{1}$$

2. *Variable substitution*; for each *diff* node, change the names of the variables which appear in the right-hand operand (the "subtrahend") but not in the left-hand (the "minuend") with new, globally unique names.
3. *Transformation*; the algebraic structure of the query is transformed, by exploiting the commutativity of $\bowtie$ and $\cup$, the distributivity of $\bowtie$ and the left distributivity of $-_{pred}$ with respect to $\cup$ and the rules listed below, until no more transformations are possible[8]:

$$\begin{aligned}
\sigma_{pred}(A \cup B) &\Rightarrow \sigma_{pred}(A) \cup \sigma_{pred}(B) && (2)\\
A -_{pred} (B \cup C) &\Rightarrow (A -_{pred} C) -_{pred} B && (3)\\
\sigma_{pred}(A) \bowtie B &\Rightarrow \sigma_{pred}(A \bowtie B) && (4)\\
\sigma_{pred_1}(A) -_{pred_2} B &\Rightarrow \sigma_{pred_1}(A -_{pred_2} B) && (5)\\
(A -_{pred} B) \bowtie C &\Rightarrow (A \bowtie C) -_{pred} B && (6)\\
BGP_1 \bowtie BGP_2 &\Rightarrow \text{merge of } BGP_1 \text{ and } BGP_2 && (7)
\end{aligned}$$

The effect of these rules is to rearrange the operators to obtain the following order (from the top) $\cup, \sigma_{pred}, -_{pred}$; note that *join* operators are all removed by rule 7. As shown in Figure 3, a normalized query consists of an (optional) union of *basic queries* each one consisting of a BGP whose results can be filtered by one or more *diff* operations. Roughly, each basic query will originate a SELECT . . . FROM . . . WHERE . . . object query with nested sub-queries for *diff* operators; the final result-set will be obtained by executing these queries and merging their results. Figure 4 (b) shows the normalized algebra for the example query.

Semantic analysis. This step aims at *transforming* the normalized query so that (1) constraints on URIs are restated in terms of constraints on the ID

[8] Rule 6 is only valid thanks to the variable substitution performed in the previous step, which avoids variable names clashes when moving up the *diff* node.

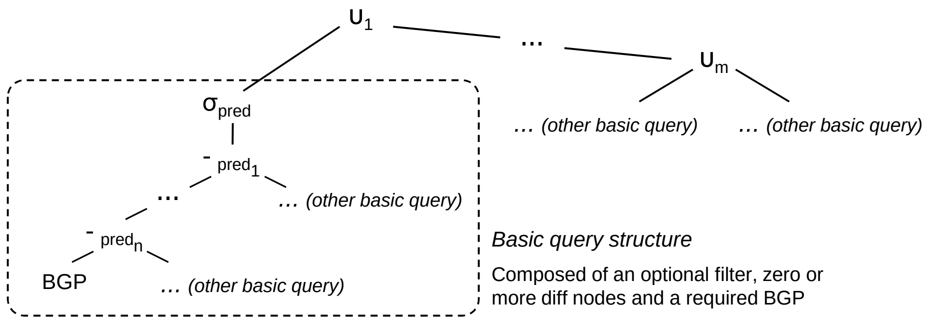


Fig. 3. Normalized query structure

attribute (2) variables on triple predicates are removed (by enumerating the possible cases) and (3) each URI or non-literal variable is associated to a single OO class. Each of these goals is addressed in a different analysis step:

1. *Rewriting of URIs*; for each URI $\langle x \rangle$ in the query, all of its occurrences are replaced with a new variable $?x$, while a new ⟨?x :hasID ID⟩ triple is added to each BGP of the query which uses the variable. Note that the ID can be extracted from the URI textual representation (see Section 3).
2. *Rewriting of variables on predicates*; each BGP containing such variables is replaced with a *union* of multiple BGPs, each one corresponding to an acceptable combination of predicate assignments to these variables. A reasoner can be used to identify the alternatives, by classifying nodes in the BGP and exploiting the domain and range constraints of predicates[9]. The assignment of predicates to variables is recorded in an auxiliary data structure for each *basic query*, in order to return them together with the results in case variables on predicates are included in the SELECT clause (or CONSTRUCT template) of the query. Finally, since the algebraic structure is modified, at the end of this step the query may need to be re-normalized again.
3. *BGP validation and class assignment.* A check is done that each variable is used only as a literal or URI, but not both. Then, a graph is built for each BGP by removing all the triples containing literal values, and a blank node is introduced for each other variable. An OWL DL reasoner is used to (1) check if this graph is consistent with the ontology and (2) infer new *rdf:type* triples for resources and variables (the blank nodes), which allow to associate an OO class to each node. If any of the checks fails, the BGP is discarded and the algebraic structure is adjusted accordingly (e.g. by removing parent *diff* or *filter* nodes too).

The query resulting from semantic analysis is ready to be translated. Figure 4 (c) shows the result of the semantic analysis for the query of the running example.

[9] The reasoner can be used as explained in step *BGP validation and class assignment*; note, however, that the choice of predicates is not critical, because even if invalid predicates are considered, the next validation step will remove them

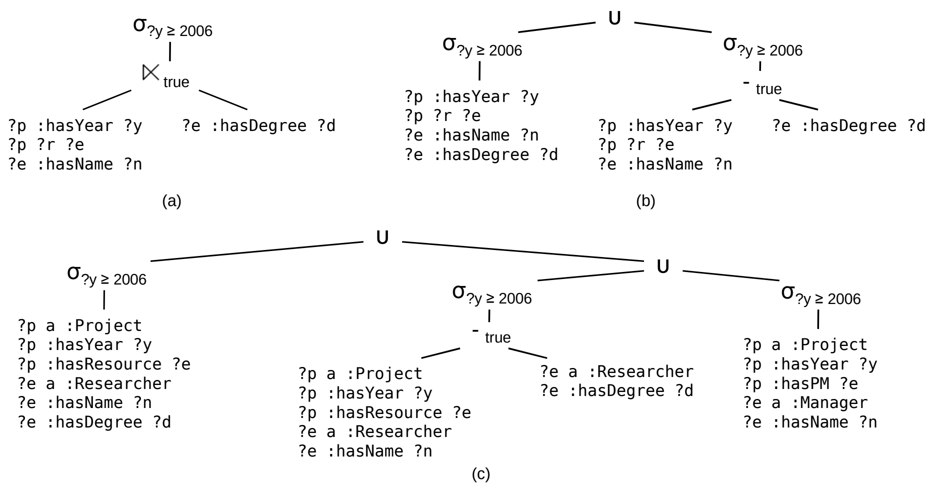


Fig. 4. Analysis of the example query: (a) parsed query, (b) normalized query (c) resulting query.

4.2 Translation

This phase is divided in two steps: translation in *monoid comprehension calculus* and normalization of the resulting expression. The first step starts translating the BGPs and then each SPARQL algebraic operator, using a *bottom-up* approach; the second step aims at reaching a normalized form of the expression, through a set of normalization rules defined in [5].

The *monoid comprehension calculus* is a framework for object query processing and optimization. We now give a brief overview of this *calculus*, readers are referred to [5] for more detailed information.

Object query languages deal with collections of homogeneous (i.e. of the same type) objects and primitive values such as *sets*, *bags* and *lists*, whose semantics can be captured by the notion of a *monoid*. A monoid is an algebraic structure consisting in a set of elements and a binary operation defined on them having particular algebraic properties. Collections of objects and operations on them (such as set and bag union and list concatenation, but also aggregate operations like max and count) can be represented as *collection monoids*; similarly, operations like conjunctions and disjunctions on booleans and integer addition over collections can also be expressed in terms of so-called *primitive monoids*.

The basic structure of the calculus is the *monoid comprehension*, that can describe a query or a part of it. This structure takes the form $\oplus\{e \mid \bar{q}\}$, where:

- $\oplus$ is a function called *accumulator*, that identifies the type of monoid by specifying how to compose (i.e. which operation should be used) the elements obtained by the evaluation of the comprehension;
- e is called *head* and it is the expression that defines the result;

– $\bar{q}$ is a sequence of *qualifiers*; these can be *generators* of the form $v \leftarrow e'$, where v is a variable ranging over the collection produced by the expression e' (which can be a monoid comprehension too), or *filters* of the form *pred*, which express constraints over the variable bindings produced by the generators.

For instance, this monoid comprehension: $\uplus\{v_1, v_2 | v_1 \leftarrow X, v_2 \leftarrow X.y, v_2 > n\}$ can be read as: "for all v_1 in X and for all v_2 in $X.y$ such that $v_2 > n$ consider the pairs v_1, v_2 and merge them (by applying the $\uplus$ accumulator) to obtain a *bag*". The accumulator functions in our translation are only $\uplus$ and $\vee$: the former defines a bag of solutions, while the latter is used to define the existential quantification.

BGP translation. A generic BGP contains a set of *triple patterns.* At the beginning of this step we reorder these triples. A set of triple patterns can be viewed as a directed graph, with vertices corresponding to subjects and objects and edges between them corresponding to triples and labelled with their predicates; if the graph contains some cycles, we break them by duplicating a vertex, thus obtaining a directed acyclic graph (DAG). To order the triples we perform a depth-first visit, starting from the root nodes of the DAG. Triples with *rdf:type* as predicate are not considered during the reordering process: they are removed and used later to resolve the assignment of variables to OO classes (as described below). Figure 5 shows the reordering process for a BGP of the running example (the leftmost in Figure 4 (c)).

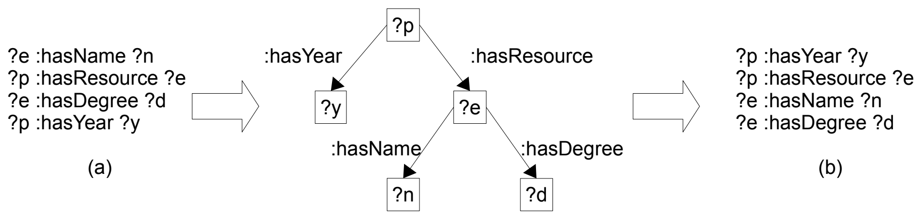


Fig. 5. Triples reordering

Now the BGP is translated in the corresponding monoid comprehension following these criteria:

1. the *accumulator* function is always $\uplus$, because a BGP returns a bag of solutions;
2. the set containing all the variables contained in the triples forms the *head* of the monoid comprehension;
3. the *qualifiers* in the *body* of the comprehension are generated by iterating over the ordered triples $\langle var_{sub}$ pred $obj\rangle$ and applying the following rules to each one:
   – **if** var_{sub} occurs for the first time, a new *generator* $var_{sub} \leftarrow Class$ (where $Class$ is the OO class assigned to the variable) is added;
   – **if** obj is a variable var_{obj} occurring for the first time, a *generator* of the form $var_{obj} \leftarrow var_{sub}.pred$ (the symbol $\leftarrow$ is changed with $\equiv$ when *pred*

is a functional property) is created. If *pred* is a functional property and var_{obj} does not appear as the *subject* of other triples, a *filter* of the form $var_{obj} \neq null$ is added too[10];

- **if** *obj* is a literal or a variable already encountered, a new *filter* is created:
  - **if** *pred* is a functional property, the *filter* takes the following form: $var_{sub}.pred = obj$;
  - **else** the *filter* takes the form: $var' \leftarrow var_{sub}.pred, var' = obj$ (where var' is a new globally unique variable).

Equation 8 shows the resulting comprehension for the BGP of Figure 5.

$$\uplus\{p, y, e, n, d \mid p \leftarrow Project, y \equiv p.year, y \neq null, e \leftarrow p.resources,\\ n \equiv e.name, n \neq null, d \equiv e.degree, d \neq null\} \qquad (8)$$

Compound constructs translation. Each SPARQL algebraic operator can be translated to a corresponding monoid comprehension expression. Using P to describe a generic *pattern* (BGPs or group-graph-patterns, i.e. BGPs composed with algebraic operators), we indicate with $\tau(P)$ the translation of P.

In Table 2 are shown the translation rules. These rules are applied using a *bottom-up* approach, starting from the leaves of the tree and moving up towards the root (see Figure 4(c)). We do not define rules for *join* ($\bowtie$) and *left join* ($\ltimes_{pred}$) because these operators are eliminated in the *analysis* step (see Section 4.1).

Rule	SPARQL algebra	Monoid Comprehension
T1	P	$\tau(P)$
T2	$\sigma_{pred}(P)$	$\uplus\{x \mid x \leftarrow \tau(P), pred\}$
T3	$\cup(P_A, P_B)$	$\tau(P_A) \uplus \tau(P_B)$
T4	$-_{pred}(P_A, P_B)$	$\uplus\{x \mid x \leftarrow \tau(P_A), \neg \vee \{pred \mid y \leftarrow \tau(P_B)\}\}$

Table 2. Translation of SPARQL Algebra constructs

Simplification rules. At the end of the translation step, we obtain a composition of nested monoid comprehensions. In their work [5], Fegaras and Maier suggest a set of *meaning-preserving* normalization rules, to unnest many kinds of nested monoid comprehension. The relevant rules for our approach are shown in Table 3.

Rule	Before	After
N1	$\oplus\{e \mid \bar{q}, v \leftarrow (e_1 \otimes e_2), \bar{s}\}$	$(\oplus\{e \mid \bar{q}, v \leftarrow e_1, \bar{s}\}) \oplus (\oplus\{e \mid \bar{q}, v \leftarrow e_2, \bar{s}\})$ for commutative $\oplus$ or empty $\bar{q}$
N2	$\oplus\{e \mid \bar{q}, v \leftarrow \otimes\{e' \mid \bar{r}\}, \bar{s}\}$	$\oplus\{e \mid \bar{q}, \bar{r}, v \equiv e', \bar{s}\}$

Table 3. Relevant normalization rules

The monoid comprehension expression resulting from the example query (Figure 4 (c)) is the following:

[10] Not null constraints are required because all variables must be bound to a value in solutions of a BGP pattern.

$$
\begin{gathered}
(\uplus\{p, y, e, n, d \mid p \leftarrow Project, y \equiv p.year, y \neq null, e \leftarrow p.resources,\\
n \equiv e.name, n \neq null, d \equiv e.degree, d \neq null, y \geq \text{“2006”}\})\\
\uplus\\
(\uplus\{p, y, e, n \mid p \leftarrow Project, y \equiv p.year, y \neq null, e \leftarrow p.resources,\\
n \equiv e.name, n \neq null, y \geq \text{“2006”}, \neg \vee \{true \mid d \equiv e.degree, d \neq null\}\})\\
\uplus\\
(\uplus\{p, y, e, n \mid p \leftarrow Project, y \equiv p.year, y \neq null, e \equiv p.pm, n \equiv e.name,\\
n \neq null, y \geq \text{“2006”}\})
\end{gathered} \tag{9}
$$

The expression obtained at the end of these steps can be already translated into object queries. However, exploiting the comprehension calculus it can be further optimized, e.g., simplifying some variables or collapsing some monoid comprehensions. We do not describe this process here due to limited space and because we are still working to identify a set of general simplification rules. To give an idea of the possible improvements, however, we show in Equation 10 the optimized expression for the example query.

$$
\begin{gathered}
(\uplus\{p, e, n, d \mid p \leftarrow Project, e \leftarrow p.resources, e.name \neq null,\\
p.year \geq \text{“2006”}, n \equiv e.name, d \equiv e.degree\})\\
\uplus\\
(\uplus\{p, e, n \mid p \leftarrow Project, e \equiv p.pm, e.name \neq null,\\
p.year \geq \text{“2006”}, n \equiv e.name\})
\end{gathered} \tag{10}
$$

4.3 Execution

In this last step we translate the normalized monoid comprehension expression into object queries, execute them on the datasource and convert the results in the format expected by the original SPARQL query. In this section we describe the translation to OQL; note, however, that the translation to other OQL dialects (such as JDOQL used by SPOON) is similar.

The normalized expression produced by the translation phase is a *union* of monoid comprehensions. Each of these monoid comprehensions is translated to a separate object query in a straightforward manner: all the expressions for the variables in the *head* are returned in the SELECT clause (for object variables we extract the object IDs, not the full objects), *generators* become the collections on which variables iterate in the FROM clause and *filters* become a conjunction of constraints in the WHERE clause. The monoid comprehension of the form: “$\neg \vee \{\ldots\}$” (that appears in rule T4 of Table 2) becomes a *subquery* of the form: “NOT EXISTS (SELECT. . .)”, also belonging to the WHERE clause.

The OQL translation of the running example query is reported below. We show the translation of the simplified comprehensions of Equation 10; however, translation to object queries is applicable starting from the comprehensions of Equation 9 (but the resulting queries would be not so compact.).

```
SELECT p.id, e.id, e.name, e.degree
FROM Projects p, p.resources e
WHERE e.name != null AND
        p.year >= 2006

SELECT p.id, p.pm.id, p.pm.name
FROM Projects p
WHERE p.pm.name != null AND
        p.year >= 2006
```

The queries obtained so far are executed one by one, then the result-sets are merged together and SPARQL *solution sequence modifiers* [3] (*order by*, *distinct*, *reduced*, *offset* and *limit*) are applied to the whole result-set. The last thing to do is the conversion of the obtained result-set in the format expected by the SPARQL query:

– for SELECT queries, we select from the result-set only the requested variables and return a *table*-form result-set;
– for ASK queries, we return *true* if the result-set is not empty, *false* otherwise;
– for CONSTRUCT queries, we create an RDF graph with the data from the result-set;
– DESCRIBE queries are currently not directly supported by our approach, however a DESCRIBE query can always be translated to a CONSTRUCT query that asks for all the triples with the desired resource as subject or object, and this kind of query is supported by our approach.

5 Implementation and evaluation

With regards to the comprehensive framework we presented in Section 4, to prove the feasibility of our approach, we implemented SPOON – SParql to Object Oriented eNgine – a tool based on Jena and JPOX which helps the automatic mapping between an OO model and the respective ontological abstraction and translates SPARQL queries in JDOQL [15] queries. The first implementation of SPOON is focused on the main constructs, namely BGP and FILTER, and it does not yet support variables on predicates.

In order to compare our approach with existing and competing systems, we chose to set up an evaluation framework, by applying different approaches to the same data source. We chose Gene Ontology data source (GO), which is available in different formats among which a SQL dump and a RDF format[11].

Our evaluation, therefore, is conducted as follows: given a SPARQL query, (1) it is translated by SPOON into JDOQL and executed by JPOX over the relational source of GO, (2) it is mapped to the GO relational database through D2R and (3) it is executed directly to the RDF version of GO loaded in a Sesame Native store (we also used the respective SQL query run on MySQL as

[11] We modified a bit the RDF version of GO available at `http://www.geneontology.org/`, because it contains some errors that make it not well-formed RDF.

a baseline reference). We stressed the system with three different queries with increasing complexity; the comparison of results is offered in Table 4, while in Table 5 we distinguish SPOON performances in *translation time* (from SPARQL to JDOQL) and *execution time* (by JPOX).

Query	**SPOON**	**D2R**	**Sesame**	MySQL
Query nr.1	291ms	695ms	280ms	95ms
Query nr.2	313ms	774ms	281ms	70ms
Query nr.3	540ms	3808ms	63620ms	179ms

Table 4. Response time of the evaluated systems with the test queries.

Query	τ	χ
Query nr.1	14ms	277ms
Query nr.2	14ms	299ms
Query nr.3	177ms	363ms

Table 5. SPOON response time divided in translation (τ) and execution (χ) time.

The recorded performances, although preliminary and partial, show an evident advantage in using our approach. A detailed report with more discussion about SPOON and its evaluation (queries, testing environment, configurations, etc.) is available at `http://swa.cefriel.it/SPOON`.

6 Conclusions

In this paper we presented our approach to the wrapping of heterogeneous data sources to expose them as SPARQL endpoints; we employ an object-oriented paradigm to abstract from the specific source format, as in ORM solutions, and we base the run-time translation of SPARQL queries into an OO query language on the correspondence between SPARQL algebra and monoid comprehension calculus. Finally, we realized a proof of concept with SPOON, which implements (a part of) our proposed framework, to evaluate it against competing approaches and we proved the effectiveness and the potentials of our approach.

Our future work will be devoted to the extension of SPOON implementation to cover other SPARQL options (like OPTIONAL and UNION); we also plan to extend the evaluation of our approach, from the point of view of the expressivity and variance of the automatic mapping between the models.

Acknowledgments

The work described in this paper is the main topic of Walter Corno's Master Thesis in Computer Engineering at Politecnico of Milano. The research has been partially supported by the NeP4B project, co-funded by the Italian Ministry of University and Research (MIUR project, FIRB-2005). We would also like to thank professor Stefano Ceri for his guidance and our colleagues at CEFRIEL for their support.

References

1. Bizer, C., Cyganiak, R., Heath, T.: How to Publish Linked Data on the Web. (2007)
2. Berrueta, D., Phipps, J.: Best Practice Recipes for Publishing RDF Vocabularies – W3C Working Draft. (2008)
3. Seaborne, A., Prud'hommeaux, E.: SPARQL Query Language for RDF – W3C Recommendation. (2008)
4. Torres, E., Feigenbaum, L., Clark, K.G.: SPARQL Protocol for RDF – W3C Recommendation. (2008)
5. Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. **25**(4) (2000) 457–516
6. Cyganiak, R.: A relational algebra for SPARQL. Technical report, HP Labs (2005)
7. D2RQ: The D2RQ Platform - Treating Non-RDF Relational Databases as Virtual RDF Graphs
8. Prud'hommeaux, E.: Adding SPARQL Support to MySQL (2006)
9. de Laborda, C.P., Conrad, S.: Relational.OWL - A Data and Schema Representation Format Based on OWL. In: Proceedings of the Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005). (2005)
10. Blakeley, C.: Virtuoso RDF Views. OpenLink Software. (2007)
11. Barrasa, J., Corcho, O., Gómez-Pérez, A.: R_2O, an Extensible and Semantically Based Database-to-ontology Mapping Language. In: Proceeding of the Second International Workshop on Semantic Web and Databases. (2004)
12. Cullot, N., Ghawi, R., Yétongnon, K.: DB2OWL: A Tool for Automatic Database-to-Ontology Mapping. Université de Bourgogne. (2007)
13. Atkinson, M., et al.: The Object-Oriented Database Manifesto. In: Proceedings of the First Intl. Conference on Deductive and Object-Oriented Databases. (1989)
14. Cattell, R., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F., eds.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (1999)
15. Russell, C.: Java Data Objects 2.0 JSR243. Sun Microsystems Inc. (2006)
16. Peim, M., Franconi, E., Paton, N.W., Goble, C.A.: Querying Objects with Description Logics
17. Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: ActiveRDF: Object-Oriented Semantic Web Programming. In: Proceedings of the Sixteenth International World Wide Web Conference. (2007)
18. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.: Automatic Mapping of OWL Ontologies into Java. In: Proceedings of the International Conference of Software Engineering and Knowledge Engineering. (2004)
19. Athanasiadis, I.N., Villa, F., Rizzoli, A.E.: Enabling knowledge-based software engineering through semantic-object-relational mappings. In: Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering. (2007)
20. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language. (2004)