Viewing the Semantic Web Through RVL Lenses*

Aimilia Magkanaraki[1], Val Tannen[2], Vassilis Christophides[1], and Dimitris Plexousakis[1]

[1] Institute of Computer Science, FORTH, Vassilika Vouton, P.O.Box 1385, GR 71110, Heraklion, Greece and
Department of Computer Science, University of Crete, GR 71409, Heraklion, Greece
{aimilia, christop, dp}@ics.forth.gr
[2] Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street Philadelphia, Pennsylvania 19104-6389
val@cis.upenn.edu

Abstract. Personalized access and content syndication involving diverse conceptual representations of information resources are two of the key challenges of real-scale Semantic Web (SW) applications, such as e-Commerce, e-Learning or e-Science portals. RDF/S represents nowadays the core SW language for creating and exchanging resource descriptions worldwide. Unfortunately, full-fledged view definition languages for the RDF/S data model addressing these challenges are still missing. We propose *RVL*, a view definition language capable of creating not only virtual resource descriptions, but also virtual RDF/S schemas from (meta)classes, properties, as well as, resource descriptions available on the Semantic Web. *RVL* exploits the functional nature and type system of the RQL query language in order to navigate, filter and restructure complex RDF/S schema and resource description graphs.

1 Introduction

The syndication and personalization of web resources, including semantic reconciliation and integration of heterogeneous metadata, are nowadays emerging as key challenges for Semantic Web [5] applications, such as e-Learning, e-Commerce or e-Science portals. Metadata provide the means to describe resources, thereby facilitating their manipulation both by applications and humans. The core Semantic Web (SW) language for creating and exchanging resource descriptions worldwide is the **Resource Description Framework/Schema Language** (RDF/S) [16, 8], which provides i) a *Standard Representation Language* [16] for metadata based on *directed labelled graphs* in which nodes are called *resources* (or *literals*) and edges are called *properties*; ii) a *Schema Definition Language* (RDFS) [8] for creating vocabularies of labels for these graph nodes (called *classes*) and edges (called *property types*); and iii) an *XML* [7] *syntax* for expressing metadata and schemas.

* This work was partially supported by the EU project SeLeNe (IST-2001-39045).

The declarative access in the metadata repository of a SW application is facilitated by RDF/S query languages, such as *RQL* [14], a typed, functional query language for uniformly navigating/filtering on RDF/S graphs at all abstraction levels (metaschema, schema and data). However, a query language is not enough. As with any query language, formulating queries on data with complex organization may require schema knowledge beyond the needs of a given application. This difficulty can be alleviated by the use of *views*, which create *virtual* schemas and resource descriptions reflecting only the users' conception of a specific application domain. In relational databases, the standard query language, SQL, serves also as a view definition language. However, for Semantic Webs (SW) represented as RDF/S graphs, a view should not be restricted to a query returning part of the SW, as *RQL* queries do. Instead, it should extend to the restructuring of class and property hierarchies, allowing the creation of new resources and property values, and even new classes and property types. To the best of our knowledge, no language for defining such views has been proposed before. In this paper we introduce **RVL** (**RDF View Language**), an expressive view definition language designed to play this role. *RVL* provides users with the ability to define a view in the same way in which they write normal RDF/S schemas and resource descriptions, defining classes and "populating" them with resources. By exploiting the *RQL* type system and the distinction of abstraction layers in an RDF/S application, *RVL* captures the desired functionality through the use of just two operators in essence treating schema creation as the instantiation of appropriate metaclasses.

The organization of the paper is as follows: Section 2 motivates the use of the *RVL* view definition language by means of an e-Learning portal example and exhibits a first sample of the functionality it supports. Section 3 unfolds the expressiveness of *RVL* by presenting the operators it specifies and their respective functionality, while Section 4 complements the presentation of *RVL* by presenting existing related approaches. Lastly, Section 5 concludes this paper by presenting directions for future work.

2 A Motivating Example

Educational portals aggregate and classify in a semantically meaningful way various online resources for different educational audiences (e.g., instructors, learners, etc.). The main resources of information in such Portals are called *learning objects* (LO) containing any kind of material (e.g., a web page, a ppt presentation, a book, a Java applet, etc.) which can be used or referenced (using, for instance, URIs [4]) during technology supported learning. To enable effective search in educational portals, LOs are described according to e-Learning metadata standards, such as IEEE/LOM, ARIADNE or IMS[3]. E-Learning schemas and LO descriptions can be easily represented in RDF/S [16, 8].

We present in Figure 1 the RDF/S description schema and base of such a portal. The upper part of the figure presents a simplified RDF/S schema for descri-

[3] http://ltsc.ieee.org/wg12/, http://ariadne.unil.ch/Metadata/, www.imsglobal.org/

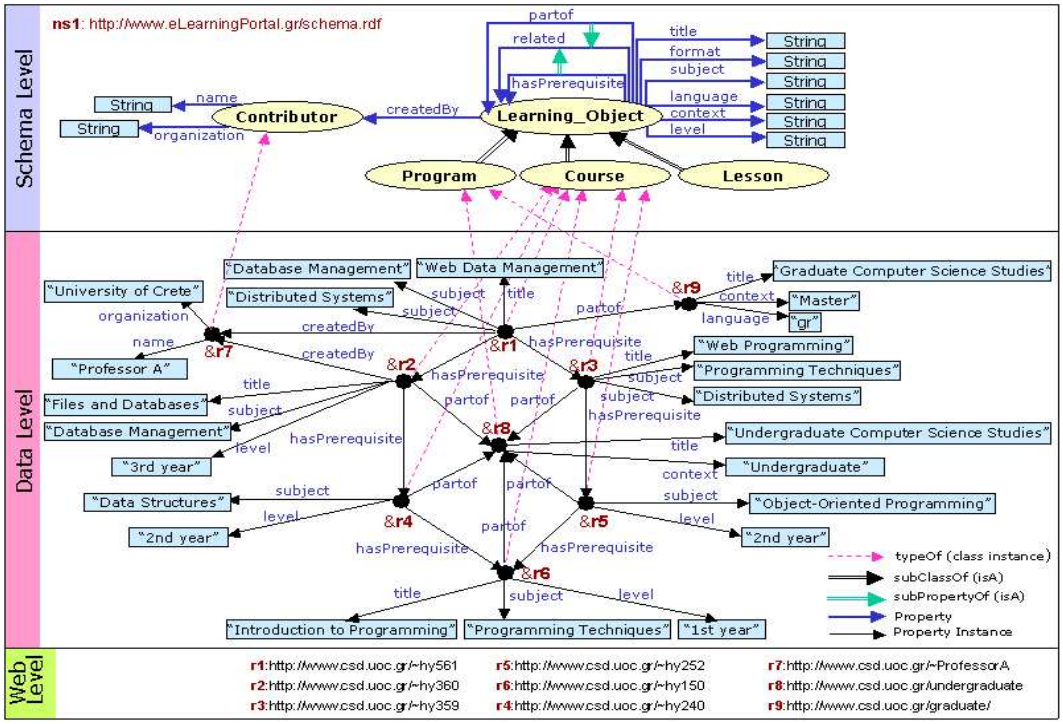

Fig. 1. An example e-Learning Portal application

bing *LO*s using attributes with information about their content (*title*, *subject*, *language*, *format* etc.), as well as their pedagogical value (educational *context* and *level*, learning objectives and time, etc.). Specialization of learning material at different granularity levels is represented by the *rdfs:subClass*-es (class subsumption) *Program*, *Course*, *Lesson* or more specific components, such as notes, assignments, exams, figures or simulation programs. Relationships between LOs like *hasPrerequisite* capturing learning dependency graphs or *partof* capturing learning material composition trees, are defined as *rdfs:subProperty*-'s (property subsumption) of the abstract relationship *related* (according to e-Learning standards other specializations are also possible). Finally, LOs may be also related to other classes of resources through relationships like *createdBy* ranging over instances of the class *Contributor*, described in turn by attributes like *name* and *organization*.

The lower part of Figure 1 illustrates the descriptions of some LOs provided by the Web site of the Computer Science Department of the University of Crete (CSD Uoc). For instance, the LO *&r1* is of *rdf:type Course* and has a *title* attribute with value "Web Data Management" and two *subject* attributes with values "Database Management" and "Distributed Systems". In addition, *&r1* is *part of* *&r9* (i.e., the graduate studies *Program* of the CSD Uoc), has two *prerequisite*s courses *&r2* (with *title* "Files and Database") and *&r3* (with *title* "Web Programming") and it has been *createdBy* the *Contributor* *&r7* with

name "Professor A" and *organization* "University of Crete". In a similar way are described the other LOs of the CSD Uoc illustrated in Figure 1.

Searching LOs in such Semantic Web portals relies on declarative query languages for RDF/S descriptions, such as *RQL* [14]. Although portals usually provide appropriate GUIs for assisting users during searching, the formulation of effective queries depends heavily on the understanding of the portal's description "schema". Such RDF/S schema graphs can be quite complex, especially when multiple schema namespaces are employed in a more or less peer-to-peer fashion to describe LOs available on the Web. Therefore, having a central access point to the wealth of LOs is a mixed blessing, if the user must be aware of too much detail in order to search the portal. To enhance the user's experience, we need the ability to personalize the way the portal can be seen, by providing simpler virtual schemas that reflect a user's perception (e.g., for instructors or learners) of the application domain. *RVL*, the view definition language we describe in this paper, provides this ability. For instance, consider a simple virtual schema (view) for instructors, which shows only database course materials and their authors. This schema can be specified with the *RVL* statements presented in the bottom-right part of Figure 2 taking as input the RDF/S description base of Figure 1. The output of these view statements is the RDF/S virtual schema and resource descriptions presented at the top-right part of Figure 2 in an XML serialization.

In RDF/S the uniqueness of (meta)schema labels and the ability to describe resources using labels from several schemas is ensured by the XML namespace facility [6]. Thus, we use in our example the *RVL* statement:

```
CREATE NAMESPACE myview=&http://www.ics.forth.gr/LO.rdf#
```

Descriptive labels are prefixed by the namespace of the schema to which they belong (e.g., *ns1#Learning_Object*), forming in this way unique URIs. This is particularly important in the open and diverse Web world and even more so when defining views, where virtual, but different, copies of old schema labels, such as class and property names, are considered.

The second *RVL* statement in our example "creates" the virtual classes `Author` and `DBCourse` and the virtual properties `creates` and `name`:

```
VIEW rdfs:Class("DBCourse"),rdfs:Class("Author"),
     rdfs:Property("creates", Author, DBCourse),
     rdfs:Property("name", Author, xsd:string);
```

where *rdfs:Class* and *rdf:Property* are two core metaclasses provided in the default RDF/S namespaces. The semantics of these namespaces along with the XML Schema datatypes is built-in in *RVL*/*RQL* and the corresponding namespace prefixes (e.g., *rdf*, *rdfs*,*xsd*) can thus be omitted, while we can use the `USING NAMESPACE` clause to declare the namespaces used in view statements. As we will see in the next section, *RVL* also provides the ability to create virtual subsumption hierarchies or even to filter/restructure existing ones.

The third *RVL* statement "populates" the virtual classes and properties defined in the view with appropriate instances copied from the source schema illustrated in Figure 1:

```
VIEW DBCourse(Y),Author(X),creates(X,Y),name(X,W)
FROM {Y;ns1:Course}createdBy{X}.ns1:name{W},
```

Fig. 2. Comparing *RQL* to *RVL*

```
        {Y}ns1:subject{Z}
    WHERE Z like "Database Management";
```

This statement works much like a query on the portal description base. In fact, to emphasize the connection we present on the left side of Figure 2 an *RQL* query that has the same `FROM` and `WHERE` clauses as the *RVL* statement. In the top-left of the figure we give the XML serialization of the result of this *RQL* query.

As we see, an *RVL* `FROM` clause consists of *RQL* [14] path expressions permitting easy navigation through complex schemas and description bases and to appropriately bind the introduced variables. Filtering conditions on these variable bindings are stated in the `WHERE` clause. For instance, the *RQL* path expression `{Y;ns1:Course}ns1:createdBy{X}.ns1:name{W}` will match all instances of class *Course* and their associated *createdBy* properties, which link them to some instance of *Contributor* and its *name* value. For each such match, we get a binding that maps Y to the *Course* resource, X to the *Contributor* and W to the *name*. In a similar way, the path expression `{Y}ns1:subject{Z}` is evaluated and the involved variable bindings are filtered according to the `WHERE` clause as well as to the implicit join condition imposed by the presence of the same variable, Y, in both path expressions. Notice however the difference between the result of the *RQL* query and the output of the *RVL* view definition in Figure 2. Although their input is the same RDF/S graph, *RVL* is capable of producing virtual schemas and resource descriptions instead of simple variable bindings represented in some (nested) tabular form.

This functionality is ensured by the `VIEW` clause, where appropriate population functions are used taking as parameters the variable bindings produced by the `FROM-WHERE` filter. For instance, the virtual class `DBCourse` is populated with instances (bound to variable Y) of the original class *Course* having a property *subject* valued "Database Management". The virtual class `Author` is populated with instances (bound to variable X) of the base class *Contributor*, which are the range values of the property *createdBy* applied to *Course* resources. In other words, `Author` is populated with all the contributors who have created a database course. Virtual properties are populated with pairs of resources (e.g., `creates` is populated with authors having created database courses) or pairs of resources-values (e.g., `name` is populated with the names of database course authors). One of the most salient *RVL* features is its ability to create virtual schemas by simply populating the two core RDF/S metaclasses *Class* (e.g., with schema classes `Author` and `DBCourse`) and *Property* (e.g., with schema properties `creates` and `name`).

For somebody interested only in database learning material, this view is much easier to understand. One can then easily formulate queries *on the view* such as the following one in *RQL*:

```
SELECT Y
FROM {X}myview:creates{Y}, {X}myview:name{Z}
WHERE Z = "Professor A"
USING NAMESPACE myview=&http://www.ics.forth.gr/LO.rdf#
```

This query should retrieve the database courses created by the author named "Professor A".

3 RVL: A View Definition Language for RDF/S

Motivated by the previous example, a fundamental question one can naturally pose, is "*what is a good specification of views for the RDF/S data model?*". We have designed *RVL* as a conceptually simple language enabling both humans and applications to understand view specifications as normal RDF/S schemas and resource descriptions. More precisely, an *RVL* view specifies a **virtual description schema** graph (or **virtual schema** for brevity). Its extension corresponds to a **virtual description base** graph (or **virtual base** for brevity), which is a valid instance of the virtual view schema. Thus, *RVL* views produce new RDF/S (meta)classes and properties which are **virtual** and their instances are computed from the **source** base(s) or schema(s) using the *RVL* program specifying the view. This program defines essentially the *mapping* (i.e., transformation) of the input (i.e., source) to the output (i.e., virtual) RDF/S graph(s).

3.1 RVL Design Choices

In order to design an effective RDF/S view specification language we have addressed the following issues:

1. How are the virtual schema (meta)classes and properties of a view related to the source description schema(s)?

2. How are the virtual base resources and property values of a view related to source description base(s)?
3. What is the expressiveness of the input/output transformations supported by the view specification language?
4. How can the output of view specifications be used in queries and other views?

In the sequel, we will present the main design choices for *RVL* in response to the above fundamental issues.

Logical data independence is one of the most important properties that a view definition language should respect (recall the ANSI-SPARC three-level architecture [3]). It essentially requires that view specifications should be independent from those of the source schemas and bases, while the semantics of existing virtual schemas should not be altered by the definition of new ones. For this reason, the scope of virtual (meta)class and property definitions is determined in *RVL* by the namespace of the view. This is particularly useful since *RVL* allows us not only to create *new* (meta)classes and properties (as in Figure 2), but also to *import* in a view existing ones from the source schemas given as input. Imported (meta)classes and properties are simply replicated in the virtual schema and do not interfere with the source ones. Moreover, as we will see in Section 3.2, virtual subsumption hierarchies (for both classes and properties) could also be defined in a view, which are not necessarily present in the source schemas. Instead of creating a global subsumption hierarchy mixing both virtual and source (meta)classes and properties, an *RVL* virtual schema refers only to the subsumption relationships explicitly established between the virtual (meta)classes and properties. The separation of virtual from source (meta)classes and properties in *RVL* leads to smaller virtual schemas easier to understand and manage.

View instantiation capabilities. Besides the population of virtual (meta)classes and properties using, for instance, *RQL* queries (see Figure 2) over the original description base (i.e., **object-preserving views**), an *RVL* virtual schema can also be instantiated in the view (i.e., **object-generating views**) specification. These instances exist only during the activation of the view and their identifiers are generated by appropriate Skolem functions. As a matter of fact, the entire virtual schema specified in a view is essentially a new instance of the default RDF/S meta-schema (class and property names are used as unique identifiers)! As we will see in Section 3.2, this functionality is also useful in cases where virtual resource descriptions may have both a dynamic part populated with resources from the original base and a static one populated exclusively at the view level. *RVL* is powerful enough to support both kinds of view instantiation, while instances of the source schemas are simply copied into the view extension, thereby acquiring a virtual hypostasis.

Transformation expressiveness is the cornerstone of the *RVL* design in order to cope with a wide range of heterogeneities found in real-scale Semantic Web applications [15, 12]. Therefore, a view specification language should provide the ability to both create (for personalization purposes) and reconciliate (for mediation purposes) quite different conceptual representations of the same

application domain. For this reason, *RQL* is equipped with *heavy data restructuring* facilities enabling users to change the abstraction level (i.e., metaschema, schema, data) in which a particular view construct is defined. As we will detail in Section 3.2, *RVL* is capable of "promoting" literals or resources of the original description base to virtual classes as well as of "demoting" metaclasses of the original description schema to virtual classes of the view. This ability is ensured by the expressiveness of the *RQL* query language to query RDF/S information at all abstraction levels and the polymorphic type system of the *RVL* population functions (i.e., the `VIEW` clause).

Closure of view language. On the one hand, one should be able to query *RVL* views, as in the case of source schemas and description bases. Since *RVL* views introduce virtual schemas, one can use their namespace to formulate *RQL* queries (see previous section) retrieving (part of) the RDF/S graph specified by the view program. On the other hand, one should be able to create views using both source and virtual schemas. We can distinguish between two levels of view specification reuse: inside a virtual schema (intra) and across (inter) virtual schemas. Intra view reuse is not supported by *RVL*, since it gives the possibility to define the extension of a virtual (meta)class based on the extension of another virtual construct of the same view. To ensure data independence and avoid cyclic declarations of virtual classes which are hard to grasp, we impose the following restriction: the `FROM` clause of *RQL* queries defining the population of the view constructs cannot refer to information (schema and data) of the view being defined. Only inter view reuse is supported by *RVL* for creating virtual (meta)classes and properties by employing other virtual schemas. This process results in a cascade of virtual schema specifications, which ensures that the constructs of a virtual schema used in the definition of another virtual schema have already been defined.

The above design decisions were taken with the objective of devising a clear and expressive RDF/S view specification language required by a large spectrum of Semantic Web applications. In the sequel, we will detail how *RVL* implements this functionality.

3.2 RVL Operators

RVL reduces the creation of virtual schemas and description bases down to the use of two operators, namely the **instantiation** and the **subsumption** operators. In order to ensure the validity of their application and infer the type of virtual constructs, the operands of the *RVL* operators must be of a specific type, which is checked during compilation w.r.t the *RQL*/*RVL* type system using the typing rules presented at Table 1 in Appendix. In addition, the presence of this type system, facilitates a more compact declaration of view statements in the sense that the type of one entity in the source schema or base can be reused as such in the view. This ability does not prohibit users to alter the type of one element using the instantiation operator, as we will subsequently see in this section.

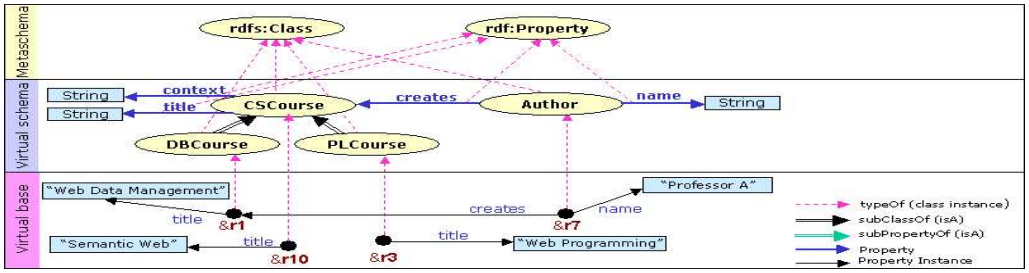

Fig. 3. A more complex *RVL* view

In the following, we will sketch out the functionality supported by each operator by using the more complex view illustrated in Figure 3. This virtual schema is defined as a view on the schema of the motivating example in Figure 1 and refers to computer science courses —especially database and programming languages courses— and their authors. In each case, we cite the typing rule of Table 1 applicable for the specific operator.

The instantiation operator, denoted "`()`", exploits the existence of abstraction layers in an RDF/S graph to support: (a) the creation/import of virtual (meta)classes and properties and (b) the population of virtual (meta)classes and properties. The instantiation of a virtual construct should be performed only with resources at the immediate lower abstraction level (see rules `9-12` in Table 1). Changing the type of an RDF/S entity in an *RVL* view compared to a source schema or base (e.g., a literal to class, or a metaclass to a class) is also supported using more complex *RVL* expressions.

Let us examine the functionality of the instantiation operator by means of the example view illustrated in Figure 3. In the simplest case, we are interested in creating new virtual classes as follows (see rule `3`):

```
VIEW Class("Author"),Class("CSCourse"),Class("DBCourse"),Class("PLCourse");
```

The first operand of "`()`" is the (meta)class (e.g., *Class*) one wants to populate with a new instance identified by the string value of the second operand (e.g., `Author`). Virtual metaclasses of classes and properties can be also created by instantiating the *RVL* built-in (meta)metaclasses *rvl:MetaClass* and *rvl:MetaProperty* (see rules `1` and `2` respectively).

In order to import a part (i.e., a *set*) of the classes defined in a source schema, we need first to use an *RQL* filter in order to identify which classes (or properties) are going to be imported into the virtual schema and then, use the instantiation operator in the `VIEW` clause as depicted by the following example (see rule `3`):

```
VIEW Class(X)
FROM Class{X}
WHERE namespace(X) = ns1 and X < ns1:Learning_Object;
```

The *RQL* path expression *Class*$\{X\}$ in the `FROM` clause introduces a variable X ranging over all classes, while the `WHERE` clause filters X bindings only to the subclasses (direct or transitive) of *Learning_Object* defined in the schema

namespace $ns1$. The instantiation operator "`()`" in the `VIEW` clause simply creates new instances of *Class* for each successful binding of class variable X. Since in this case we are importing in the virtual schema classes as provided by the source schema, we can omit the explicit call to the instantiation operator by just writing `VIEW X`.

This abbreviation cannot be used when we transform ("promote" or "demote") the abstraction level (i.e., metaschema, schema, data) of constructs specified in the view w.r.t. their level in the source schema and base. Assuming, for instance, that the values of the property *subject* are not simple strings but terms from a structured vocabulary (e.g., ACM Computing Classification System[4]), one can easily create virtual classes from these values using the following *RVL* statement (see rule `3`):

```
VIEW Class(X)
FROM ns1:subject{X};
```

In this example, string values will be used as unique names of the so created virtual classes. For this purpose, the instantiation operator uses appropriate Skolem functions: for two equal *subject* values, only one virtual class is created. This ability offers a great flexibility in view specification, especially in environments with high diverse modelling of resource descriptions.

As far as **properties** are concerned, *RVL* follows the RDF/S approach to consider properties as first-class citizens. Thereby, their definition is independent of the definition of the class they are attributed to, while they can be specialized forming subsumption hierarchies. The restriction posed by the *RQL*/*RVL* data model is that the domain and range of a property must always be defined and be unique, thus the creation of a (virtual) property is accompanied with the definition of its domain and range classes (or metaclasses or literal types). To accommodate for this peculiarity, the instantiation operator has a slightly different syntax. The first operand of the instantiation operator corresponds to the name of the core metaclass of properties (*Property*), the second to the name of the virtual property, the third to its domain and the fourth to its range. In the simplest case, we are interested in creating new virtual properties as follows (see rule `4`):

```
VIEW Property("creates", Author, CSCourse),
     Property("name", Author, xsd:string),
     Property("context", CSCourse, xsd:string),
     Property("title", CSCourse, xsd:string);
```

This view statement creates four new instances of the metaclass *Property* uniquely identified by their names: the virtual property `creates` emanating from the virtual class `Author` and ranging over the virtual class `CSCourse` as well as the virtual attributes `name`, `context` and `title` of type `string` having as domain respectively the virtual class `Author` and `CSCourse`.

Due to the functional nature of *RVL*, the operands of the instantiation operators could be not only atoms (constants or variables) but also other *RVL*/*RQL* expressions of an appropriate type. For instance, we could define *inverse* properties using the following *RVL* statement (see rule `4`):

```
VIEW Property("creator",domain(ns1:createdBy),range(ns1:createdBy));
```

[4] http://www.acm.org/class/1998/

In this example, the virtual property `creator` is created with domain and range the virtual classes `Contributor` and `Learning_Object` respectively returned by the employed *RQL* functions. This is an example of another possible *RVL* abbreviated expression: the domain and range virtual classes `Contributor` and `Learning_Object` are defined in the view at the same time as the property `creator`. The complete syntax of the `VIEW` clause comprises the expressions: `Class(domain(ns1:createdBy))` and `Class(range(ns1:createdBy))`.

As in the case of classes, we can import in the view a part (i.e., a *set*) of the properties defined in a source schema as follows (rule `4`):

```
VIEW Property(P, CSCourse, range(P))
FROM Property{P}
WHERE domain(P)=ns1:Learning_Object and P < ns1:related;
```

According to our example of Figure 1, this *RVL* statement creates two instances of the metaclass *Property* with names `partof` and `hasPrerequisite` with domain the already defined virtual class `CSCourse` and with the same ranges as in the source schema identified by the namespace `ns1`.

Besides creating virtual schemas we also need to populate the virtual classes and properties specified in the view. The same instantiation operator is used for this purpose taking this time operands of different types. The additional restriction imposed in the case of properties is that the resources at the data level to which a property is attributed are instances of the domain and range classes of the property at schema level. The following two *RVL* statements populate the virtual classes and properties we defined above for the example of Figure 3 (see rules `11` and `12` respectively):

```
VIEW DBCourse(Y),creates(X,Y),Author(X),name(X,W),context(Y,Z),title(Y,K)
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W}, {Y}ns1:context{Z},
     {Y}ns1:title{K}, {Y}ns1:subject{L}
WHERE L like "Database Management";


VIEW PLCourse(Y),creates(X,Y),Author(X),name(X,W),context(Y,Z),title(Y,K)
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W}, {Y}ns1:context{Z},
     {Y}ns1:title{K}, {Y}ns1:subject{L}
WHERE L like "*Programming*";
```

The virtual class `DBCourse` (`PLCourse`) is populated with instances of the source class *Course* having a property *subject* valued "Database Management" ("Programming Techniques" or "Object-Oriented Programming"). The virtual class `Author` is populated in both cases by *Contributor* instances having created (property *createdBy*) *Course* instances on the desired *subject*. Virtual properties are populated in a similar way (`DBCourse` and `PLCourse` are defined as subclasses of `CSCourse` in the next section).

As a last example we illustrate how virtual classes (or properties) can be populated with virtual resources residing exclusively at the view. Assuming that an instructor wants also to include within the virtual base `CSCourse`s published by himself, he/she can issue the following *RVL* statement (rules `11` and `12`):

```
VIEW CSCourse(&http://www.mycourses.net/~SemWeb),
     title(&http://www.mycourses.net/~SemWeb, "Semantic Web");
```

As we will see in the next subsection, by defining `DBCourse` and `PLCourse` as subclasses of `CSCourse`, the final population of `CSCourse` will contain its proper instances, as well as, those of its subclasses.

In more complex situations, an instructor may want to populate the `DBCourse` virtual class with resources from a source base, while complete their description manually, by adding, for instance, a learning `objective` property:

```
VIEW DBCourse(X),objective(X,"research tutorial")
FROM {X;ns1:Course}ns1:subject{Y},
WHERE Y like "Database Management";
```

The above *RVL* statement will create for each LO instance of `DBCourse` an `objective` property with value "research tutorial" (the property is assumed to be already defined in the view).

The subsumption operator, denoted "`< >`", is mainly used for defining virtual sub-(meta)classes or subproperties. Some restrictions are imposed on the use of this operator by the *RQL*/*RVL* data model. First, cycles in virtual class (or property) subsumption hierarchies are not allowed. Second, the domain and range of a property must be subsumed by the domain and range of its super properties. In addition, the subsumption operator is applicable on operands of the same type ((meta)/class and property types), since the formulation of hierarchies between entities of different type is meaningless (see rules `5-8` in Table 1).

In the simplest case one wants to explicitly define the subsumption relationship between two virtual (meta)classes or properties, as for instance in the following *RVL* statements:

```
VIEW CSCourse<DBCourse>;
VIEW CSCourse<PLCourse>;
```

The second operand (e.g., `DBCourse`) of "`< >`" is declared to be a subclass (or a subproperty) of the first one (e.g., `CSCourse`). Both operands in this example are of type class (see rule `7`).

As we have seen in the previous subsection, *RVL* give us the ability to import a part of the source schema into the view. Using the subsumption operator in conjunction with *RQL* filters, we are able to import not only the source classes (or property) names but entire subsumption hierarchies from a source schema as depicted in the following example:

```
VIEW $X<$Y>
FROM $X{;$Y}
WHERE namespace($X)=&www.eLearningPortal.gr/schema.rdf# and
      namespace($Y)=&www.eLearningPortal.gr/schema.rdf#;
```

The *RQL* path expression in the `FROM` clause essentially traverse the class subsumption hierarchy of the source schema identified by the namespace www.eLearningPortal.gr/schema.rdf. Then, for each binding of the class variable $\$X$ (e.g., to *Learning_Object*), the variable $\$Y$ is bind to the corresponding (direct of transitive) subclasses (e.g., to *Course*). The result of the original *RQL* query produces essentially a Cartesian product of each class with its subclasses. The use of the subsumption operator in the `VIEW` clause with operands the variables $\$X$ and $\$Y$ results in the reconstruction in the view of the original subsumption

hierarchy of the source schema. It should be stressed that the above *RQL* path expression considers a complete transitive closure of the subsumption hierarchy (i.e., there are all the paths from a node to its ancestors up to the root). This is extremely useful when filtering conditions on class (or property) names are also used in the `WHERE` clause. For instance, the exclusion from the view of some source classes (e.g., *Program*) results into a "connected" hierarchy relating through subsumption subclasses (e.g., figures, exams, etc) to their ancestors(s) (e.g., *Learning_Object*). Since the use of appropriate labelling schemes for class (or property) DAGS [9] alleviates the need for actually computing the transitive closure, the subsumption operator can easily produce a *minimal* form in which redundant relationships are removed.

The *RVL* examples presented in this section were just indicative of *RVL*'s expressiveness. Consider the spectrum of possible views which can be defined by changing the operands of the subsumption and instantiation operators and by exploiting the querying capabilities of *RQL*. This expressiveness allows us to think of *RVL* as a powerful transformation mechanism for RDF/S schema and resource description graphs. In addition, *RVL* allows to capture in a view several modelling constructs recently proposed in OWL [10] such as inverse properties, synonyms of classes and properties or complex class definitions using boolean expressions and existential/universal quantifiers (supported by *RQL* filters).

4 Related Work

Several view specification languages have been proposed in the database literature. The most relevant to *RVL* is work conducted in the context of ODMG-compliant object-oriented DBMS, such as O_2 [1, 18], MultiView [17], Chimera [13] and K2 [19]. These view specification languages extend the relational approach for defining views as "named queries" with features for creating virtual object schemas. Apart from the differences between the ODMG and RDF/S data models (e.g., sub-properties, multiple classification of objects, etc.) or between the underlying design choices (e.g., in transformation expressiveness), the main novelty of *RVL* compared to these languages lies in its flexibility to create virtual classes (or properties) using *RQL* queries. This functionality is particularly useful for Semantic Web applications managing large schemas in a peer-to-peer way.

Some view specification languages have also been proposed for the RDF/S data model. In [20] set-based operations have been introduced in order to define object-preserving views using an untyped version of *RQL*. Opposite to *RVL*, the logical data independence of views is violated by this language, since virtual and source classes are merged into one global schema, while restructuring constructs for subsumption hierarchies are not supported. An alternative approach has been proposed in [11], which relies on F-logic rules to define only virtual description bases. Unlike *RVL*, this language does not provide the means to define virtual RDF/S schema graphs using, for instance, meta-schema instantiation capabilities. In the same spirit, [2] proposes a variation of *RQL* in order to produce as a query result an output RDF resource description graph instead of variable

bindings in some tabular form. To the best of our knowledge *RVL* is the first language offering a full-fledged view specification for the RDF/S model.

5 Summary and Future Work

We have presented *RVL*, a language that brings a new kind of capability to the management of RDF/S metadata: users can create virtual schemas and resource descriptions customized to the needs of specific applications. By distinguishing the abstraction layers in an RDF/S application and by exploiting the *RQL* type system, *RVL* realizes the virtual schema creation as the instantiation of appropriate metaclasses and achieves its target functionality through the use of only two operators: the *instantiation* and the *subsumption* operators.

There remain several open issues to deal with in order to fully support a view definition mechanism for RDF/S. One of them is the *composition* of queries formulated against a view with the definition of the view in order to produce queries against the original RDF/S data that can be actually evaluated (thus avoiding the computation of the view data in its entirety). In relational databases composing SQL queries with SQL view definitions is fairly straightforward. Composing *RQL* queries with *RVL* views is more challenging and is a research target for us. Another important issue is checking the *consistency* of view definitions, i.e., checking whether the graph they produce satisfies the constraints of our model. Again, we wish to develop methods for consistency checking that avoid the naive approach in which the entire view data is constructed and then validated. Lastly, although we have argued for the benefits of defining virtual views, it is possible to implement an *RVL* engine that would actually compute and *materialize* the views. Such a capability would be of interest in metadata transformation applications where, for example, subsidiary but independently functioning portals are created from a given central one. This raises the classical problem of maintenance/update of materialized views, a complex problem long pondered upon by the database community. In the context of RDF/S, this problem is even more interesting, due to the peculiarities of the data model.

References

1. Abiteboul, S., Bonner, A.: Objects and Views. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, Colorado (1991) 238–247
2. Aidministrator Nederland bv: SeRQL user manual, Version:0.4 (2003), http://sesame.aidministrator.nl/publications/SeRQL
3. ANSI/X3/SPARC Study Group on Database Management Systems. Interim Report. ACM SIGMOD Bulletin 7, N2 (1975)
4. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, http://www.ietf.org/rfc/rfc2396.txt
5. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. In: Scientific American (May, 2001), http://www.sciam.com/2001/0501issue/0501berners-lee.html
6. Bray, T., Hollander, D., Layman, A.: Namespaces in XML. W3C Recommendation (1999)

7. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation (2000)
8. Brickley, D., Guha, R.V.: Resource Description Framework Schema (RDF/S) Specification 1.0. W3C Candidate Recommendation (2000)
9. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On Labeling Schemes for the Semantic Web. In: Proceedings of the 12th International World Wide Web Conference (WWW'03), Budapest, Hungary (2003)
10. Dean, M., Connolly, D., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A.: OWL Web Ontology Language 1.0 Reference. W3C Working Draft (2002)
11. Decker, S., Sintek, M., Nejdl, W.: TRIPLE: A Logic for Reasoning with Parameterized Views over Semi-Structured Data. Technical Report (2002), http://www.kbs.uni-hannover.de/Arbeiten/Publikationen/2002/triple_views.pdf
12. Durand, D., Saton, P.: Semantic Heterogeneity Among Document Encoding Schemes. Final Report for NIST Federal Assistance Contract 60NANB0D0115 (2002), http://www.stg.brown.edu/projects/semantic/semantic_stg.pdf
13. Guerrini, G., Bertino, E., Catania, B., Garcia-Molina, J.: A Formal Model of Views for Object-Oriented Database Systems. Theory and Practice of Object Systems, 3(**3**) (1997) 157–183
14. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proceedings of the Eleventh International World Wide Web Conference 2002, Honolulu, Hawaii, USA (2002)
15. Klein, M.: Combining and Relating Ontologies: An Analysis of Problems and Solutions. In: Proceedings of the IJCAI'01 Workshop on Ontologies and Information Sharing, Seattle, USA (2001)
16. Lassila, O., Swick, R.: Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation (1999)
17. Rundensteiner, E.: MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases. In: Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, Canada (1992) 187–198
18. Souza dos Santos, C., Abiteboul, S., Delobel, C.: Virtual Schemas and Bases. In: M. Jarke, J. Bubenko and K. Jeffery (editors): Proceedings of the Fourth International Conference on Extending Database Technology, St John's College, Cambridge, UK. Lecture Notes in Computer Science No. 779 (1994) 81–94
19. Tannen, V., Davidson, S.: The Information Integration System K2. In preparation.
20. Volz, R., Oberle, D., Studer, R.: Views for light-weight web ontologies. In: Proceedings of the ACM Symposium on Applied Computing SAC 2003, Melbourne, Florida, USA (2003)

Appendix: *RVL* Typing Rules

The type system foreseen by *RQL* [14] specifies a set of types, namely the `metaclass of classes` (MC) (τ_{M_c}), `metaclass of properties` (MP) (τ_{M_p}), `class` (τ_C), `property` ($\tau_P[\tau,\tau]$), `resource URIs` (τ_U), `literal` (τ_L) (XML Schema data types), `bag` ($\{.\}$), `sequence` ($[.]$) and `alternative` ($(.)$) types. The notation $\tau_P[\tau,\tau]$ for property types indicates the exact type of its domain (metaclass and class types) and range (metaclass, class and literal types) (first and second position in the sequence). For brevity, we use the notation τ_P for property types.

RVL extends this type system by specifying two more metaschema types, ω_C and ω_P, used by the instantiation operator to created user-defined metaclasses of classes and properties respectively. The restrictions and inferences specified by *RVL* are captured by the typing rules presented in Table 1. Each rule represents the drawing of a conclusion (the part below the horizontal line) on the basis of a premise (the part above the horizontal line). For instance, rule `12` states that: "*If e is an expression of property type and e_1 and e_2 are expressions of types τ_1 (resource, class or property) and τ_2 (resource, class, property or literal) respectively, then $e(e_1,e_2)$ is a valid expression of type sequence of types τ_1 and τ_2. Otherwise, a type error is returned*".

Table 1. *RVL* Typing rules

Operation	Typing Rule	
`MC creation`	$\frac{e_1:\omega_C, e_2:\tau, \tau\in\{string,\tau_{M_c},\tau_{M_p},\tau_C,\tau_P,\tau_U\}}{e_1(e_2):\tau_{M_c}}$	(1)
`MP creation`	$\frac{e_1:\omega_P, e_2:\tau, \tau\in\{string,\tau_{M_c},\tau_{M_p},\tau_C,\tau_P,\tau_U\}}{e_1(e_2):\tau_{M_p}}$	(2)
`Class creation`	$\frac{e_1:\tau_{M_c}, e_2:\tau, \tau\in\{string,\tau_{M_c},\tau_{M_p},\tau_C,\tau_P,\tau_U\}}{e_1(e_2):\tau_C}$	(3)
`Property Creation`	$\frac{e:\tau_{M_p}, e_1:\tau_1, \tau_1\in\{string,\tau_{M_c},\tau_{M_p},\tau_C,\tau_P\}\ e_2:\tau_2,\tau_2\in\{\tau_{M_c},\tau_{M_p},\tau_C\}, e_3:\tau_3,\tau_3\in\{\tau_{M_c},\tau_{M_p},\tau_C,\tau_L\}}{e(e_1,e_2,e_3):\tau_P[\tau_2,\tau_3]}$	(4)
`MC subsumption`	$\frac{e_1:\tau_{M_c}, e_2:\tau_{M_c}}{e_1 < e_2 >:[\tau_{M_c},\tau_{M_c}]}$	(5)
`MP subsumption`	$\frac{e_1:\tau_{M_p}, e_2:\tau_{M_p}}{e_1 < e_2 >:[\tau_{M_p},\tau_{M_p}]}$	(6)
`Class subsumption`	$\frac{e_1:\tau_C, e_2:\tau_C}{e_1 < e_2 >:[\tau_C,\tau_C]}$	(7)
`Property Subsumption`	$\frac{e_1:\tau_P, e_2:\tau_P}{e_1 < e_2 >:[\tau_P,\tau_P]}$	(8)
`MC population`	$\frac{e_1:\tau_{M_c}, e_2:\tau_C}{e_1(e_2):\tau_C}$	(9)
`MP population`	$\frac{e_1:\tau_{M_p}, e_2:\tau_P}{e_1(e_2):\tau_P}$	(10)
`Class population`	$\frac{e_1:\tau_C, e_2:\tau_U}{e_1(e_2):\tau_U}$	(11)
`Property population`	$\frac{e:\tau_P, e_1:\tau_1, \tau_1\in\{\tau_U,\tau_C,\tau_P\}, e_2:\tau_2, \tau_2\in\{\tau_U,\tau_C,\tau_P,\tau_L\}}{e(e_1,e_2):[\tau_1,\tau_2]}$	(12)