Available online at www.sciencedirect.com
SCIENCE DIRECT®
Web Semantics: Science, Services and Agents on the World Wide Web 1 (2004) 359–375
JOURNAL OF Web Semantics
www.elsevier.com/locate/websem

Viewing the semantic web through RVL lenses

Aimilia Magkanaraki[a,b], Val Tannen[c], Vassilis Christophides[a,b], Dimitris Plexousakis[a,b,*]

[a] *Institute of Computer Science, FORTH, Vassilika Vouton, P.O. Box 1385, GR 71110 Heraklion, Greece*
[b] *Department of Computer Science, University of Crete, P.O. Box 2208, GR 71409, Heraklion, Greece*
[c] *Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street, Pennsylvania, PA 19104-6989, USA*

Received 11 March 2004; received in revised form 28 June 2004; accepted 28 June 2004

Abstract

Personalized access and content syndication involving diverse conceptual representations of information resources are two of the key challenges of Semantic Web (SW) applications. In this paper, we investigate how these goals can be achieved using view definition primitives over RDF/S bases. We introduce a full-fledged view definition language, RVL, for creating virtual RDF/S resource descriptions and schemas from (meta)classes, properties, and resource descriptions, and we illustrate how RVL views can be composed with structured RDF/S queries expressed in a query language like RQL, by means of an internal logical framework capturing the semantics of the RDF/S model, RQL queries and RVL views.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Semantic web; Query languages; Views

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 [6] applications. For instance, e-Learning applications aim to facilitate the access to digital material—not necessarily primarily produced for educational purposes—to a wide audience of learners and instructors with diverse educational background, needs or expectations by exploiting the existence of appropriate metadata. Metadata provide the means to describe information 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) [19,10].

* Corresponding author. Tel.: +30 2810 3916 37; fax: +30 2810 3916 38.
E-mail addresses: aimilia@ics.forth.gr (A. Magkanaraki), val@ics.forth.gr (V. Tannen), christop@ics.forth.gr (V. Christophides), dp@ics.forth.gr (D. Plexousakis).

1570-8268/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.websem.2004.06.004

RDF/S (a) enables a *modular design* of descriptive schemas based on the mechanism of namespaces [9]; (b) allows easy *reuse* or *refinement* of existing schemas through *subsumption* of both class and property definitions; (c) supports partial descriptions since *properties* associated with a resource are by default *optional and repeated*; and (d) permits *super-imposed descriptions* in the sense that a resource may be multiply classified under several classes from one or several schemas. These modelling primitives are crucial for real-scale Semantic Web applications, such as e-Learning, where monolithic schemas and descriptions cannot be constructed in advance as application designers and providers cannot always cater for all possible demand scenarios and users may only have incomplete information about available resources.

The declarative access to the resource descriptions of a SW application can be easily implemented using a declarative RDF/S query language, such as **RQL** [17]. RQL is a typed, functional query language for uniformly performing navigation and filtering on RDF/S graphs at all abstraction levels (metaschema, schema and data). However, an RDF/S query language is not enough. What is useful is not just the retrieval of (part of) an RDF/S graph (as in the case of RQL), but the ability to *create* virtual (meta)schemas and resource descriptions in the result. As a matter of fact, for semantic webs represented as RDF/S graphs, a view should enable the restructuring of the employed class and property hierarchies, as well as, the creation of new resources and class or property types. This gives the ability to users to access existing resources according to their own terminology/conceptualization, i.e., the ability to use *subjective* rather than *objective* ontologies [15]. An objective ontology (e.g., IEEE LOM for educational metadata) is used by portal designers to describe resources (e.g., Learning Objects) for quite heterogeneous audiences, while a subjective ontology models the kind of descriptive terms users (e.g., instructors or learners) typically employ to describe a particular domain of interest.

The aim of this paper is to discuss the language primitives needed for defining declarative views over distributed, autonomous RDF bases holding resource descriptions and schemas. Based on these primitives, we introduce a full-fledged view definition language, called RVL, in which one can write views as normal RDF/S schemas and resource descriptions. By exploiting the RQL type system, RVL captures the desired functionality with the use of just two operators. To the best of our knowledge, no language for defining such views has been proposed yet.

The organization of the paper is as follows: Section 2 motivates the use of RVL views by means of an e-Learning portal example and exhibits a first sample of the functionality it supports. The design choices, which influenced RVL's specification, are presented in Section 3, along with the operators it specifies and their respective functionality. The composition of RVL views with structured queries formulated in RQL is exhibited through an internal logical framework that captures the RDF/S semantics and queries. Sections 4 and 5 complement the presentation of RVL by presenting existing related approaches in the field of relational, object-oriented and XML databases and conclude this paper by summarizing its contribution and drawing directions for further research.

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 [5]) during technology-supported learning. In order to enable effective search for LOs in educational portals, LO descriptions conform to e-Learning standards, such as IEEE/LOM (Learning Object Metadata), ARIADNE or IMS[1], and also to topic-specific taxonomies of scientific domains, such as ACM/CSS (Computing Classification System) [2] or taxonomies of detailed learning objectives [8]. LO schemas and descriptions can be effectively represented in the Resource Description Framework/Schema Language (RDF/S), offering advanced modelling primitives for an educational information space. Fig. 1 presents the RDF/S description schema and base of an hypothetical educational portal.

[1] , http://ariadne.unil.ch/Metadata/, www.ims-global.org/.

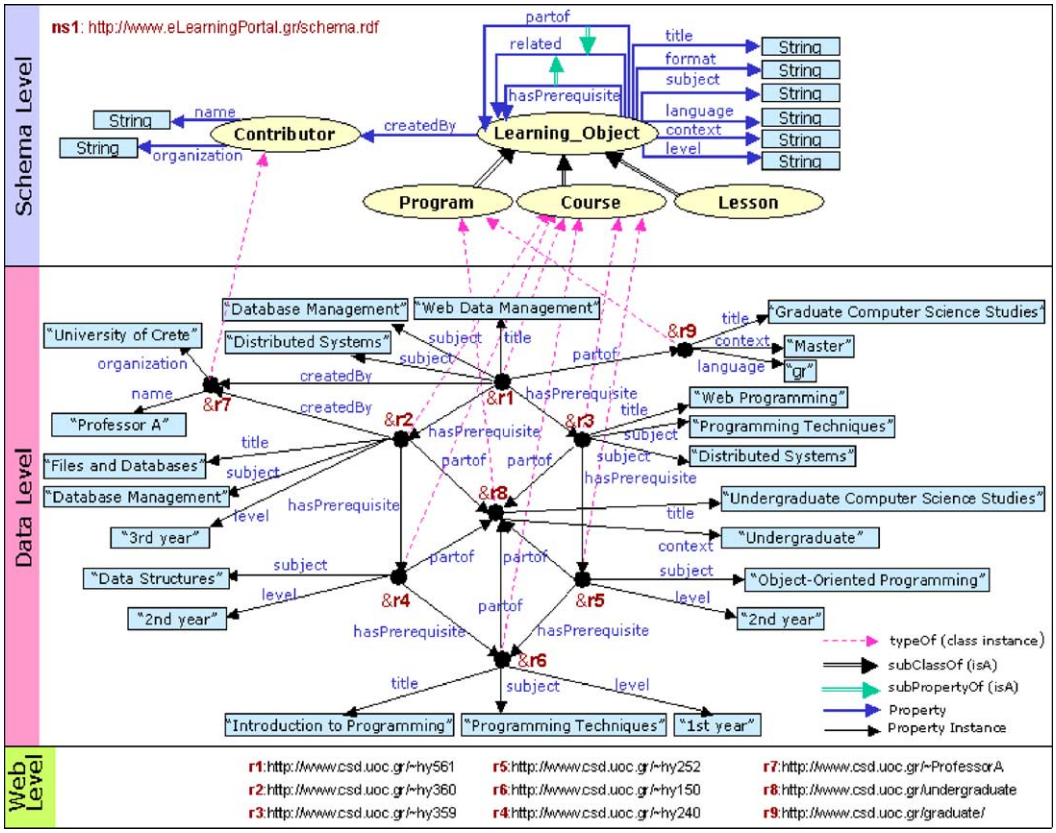


Fig. 1. An example e-Learning Portal application.

The upper part of the figure depicts a simplified RDF/S schema for describing LOs using attributes with information about their content (*title*, *subject*, *language*, *format*, etc.) and pedagogical value (educational *context* and *level*, learning objectives and time required for fulfilling an educational unit, etc.). The distinction of the different granularity levels of learning material is represented by the *rdfs:subClass-es* (class subsumption) *Program*, *Course*, *Lesson* or more specific components, such as notes, assignments, exams, figures and 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*. LOs may also be related to other classes of resources through relationships like *createdBy* ranging over instances of the class *Contributor*, which are described in turn by attributes like *name* and *organization*. These specializations and relationships among LOs are just indicative of the different property types specified in e-Learning standards [23] and provide the basic descriptive information needed for effectively querying e-Learning information.

The lower part of Fig. 1 illustrates how some LOs provided by the web site of the Computer Science Department of the University of Crete (CSD UoC) are described according to the schema employed by the educational portal of our example. For instance, the LO &*r1*, representing the web site of the "Web Data Managementcourse, is of *rdf:type Course* and has a *title* attribute valued "Web Data Management" and two *subject* attributes valued "Database Management" and "Distributed Systems" respectively. In addition, *Course* &*r1* is *part of Program* &*r9* (i.e., the graduate studies *Program* of the CSD UoC), has two *prerequisite* 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". The other LOs illustrated in the lower part of Fig. 1 are described in a similar way.

2.1. Querying Semantic Web resources with RQL

Finding LOs in an educational portal relies on declarative query languages over RDF/S descriptions, such as RQL (for an exhaustive comparison of RDF/S query languages readers are referred to [20]). In the context of the exemplary educational portal, RQL offers browsing and querying facilities over the LO information space, i.e., the LOs descriptions and the schemas they conform to. More specifically, RQL is a typed query language relying on a functional approach. The type system foreseen by RQL specifies a set of types, namely the `metaclass of classes, metaclass of properties, class, property, resource URIs`, literal (XML Schema data types), bag, sequence and alternative types. On the whole, these types ensure that RQL, due to its functional nature, can compose basic queries and iterators into complex queries.

More generally, placed in a semi-structured context, RQL treats RDF/S description graphs as a collection of nodes and edges. Schema nodes and edges can be queried as normal data using metaclass names, which essentially serve as entry-points to the corresponding graph. Furthermore, RQL supports SQL-like filters, which use generalized path expressions with variables on nodes and edges to traverse RDF/S description graphs at arbitrary depths. Thus, the `SELECT-FROM-WHERE` filters provide a powerful tool to iterate over collections with RDF data or schema information of any kind. The `SELECT` clause, as in SQL, defines a projection over the variables whose values participate in the result. The `FROM` clause hosts the defined path expressions, which essentially define the part of the RDF/S graph that will participate in the evaluation of the query. In fact, a path expression consists of a series of steps. Each step represents movement in a particular direction by identifying node labels, and each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. These filtering conditions are declared at the (optional) `WHERE` clause. The result of each step is a list of nodes that serves as a starting point for the next step. Moreover, 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. Thus, RQL uses the (optional) clause `USING NAMESPACE` for the definition of namespace prefixes.

For instance, consider the RQL query presented on the left side of Fig. 2, which retrieves all *Course* resources of *subject* "Database Management" that have been *createdBy* by a resource with a *name* attribute:

```
SELECT Y, X, W
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W},
     {Y}ns1:subject{Z}
WHERE Z like "Database Management"
USING NAMESPACE ns1=&wwww.eLearningPortal.gr/schema.rdf#
```

As we see, an `RQL FROM` clause consists of path expressions, which facilitate the navigation through complex schemas and description bases and 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* value. 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. The XML serialization of the result of this query is given in the top-left of Fig. 2.

2.2. Defining RVL views on Semantic Web resources

In addition to advanced querying facilities provided by an expressive RDF/S query language such as RQL, also needed is the personalisation of resource descriptions and schemas employed by a portal. For instance, a learner using an educational portal might want LOs presented according to his/her educational level (e.g., the postgraduate courses) and program (e.g., the e-commerce courses) and not according to the descriptive terms used by the portal designers. In this case, the need for a *subjective* rather than an *objective* ontology emerges. An objective ontology (e.g., IEEE LOM) is used by portal designers to describe resources for quite



Fig. 2. Comparing RQL to RVL.

heterogeneous educational audiences. A subjective ontology models the kind of descriptive terms instructors or learners typically employ to describe a particular domain of interest, thus becoming a classification mechanism of information resources—not necessarily primarily produced for educative purposes—to learning-specific ontologies.

Thus, to enhance the user's experience, we need the ability to personalize the way the portal can be viewed, by providing simpler virtual schemas that reflect an instructor's or learner's perception of a domain of interest. *RVL*, the view definition language considered in this paper, provides this ability by offering techniques for the reconciliation and integration of metadata describing heterogeneous distributed LOs and for the definition of personalized views over the LO information space.

To illustrate the functionality of RVL, consider a simple virtual schema (view) for instructors, which represents only database course material and its authors. This schema can be specified with the RVL statements presented in the bottom-right part of Fig. 2 taking as input the RDF/S description base of Fig. 1. The output of these view statements is the RDF/S virtual schema and resource descriptions presented at the top-right part of Fig. 2 in an XML serialization.

Since 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, in our example we use 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"),
     rdf:Property("creates", Author, DBCourse),
     rdf: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.[2] Furthermore, 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 description base illustrated in Fig. 1:

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

This statement works much like the query on the portal description base presented in Section 2.1. To emphasize the connection, we juxtapose on Fig. 2 the RVL statement and the RQL query having the same `FROM` and `WHERE` clauses. In the top row of Fig. 2, we give the XML serialization of the results of both the query and the view definition statements.

As we can observe, an RVL `FROM` clause consists of the RQL path expressions facilitating navigation through complex schemas and description bases and binding of introduced variables. Filtering conditions on these variable bindings are stated in the `WHERE` clause, as in RQL queries. Notice however the difference between the result of the RQL query and the output of the RVL view definition in Fig. 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 someone 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".

In the context of an educational portal, a view definition language such as RVL can be used to implement advanced user aids, such as personalised *navigation* and *knowledge maps*. In the first case, using appropriate GUIs, one can easily implement 'smart' bookmarks on available LOs, whose access implies a sequence of navigation steps and/or complex filtering conditions on the underlying ontology (or ontologies) of an educational portal. To use this bookmark and access its relevant LOs we just need to activate the corresponding view. Moreover, a user can further describe a smart bookmark using properties from another schema, such as the educational objective of the contained LOs, etc. In the second case, knowledge maps define and combine neighboring semantic domains for portal sub-communities. Such knowledge maps essentially enable the creation of personalised semantic portals by clustering in a meaningful way distributed description bases relevant to the educational needs and interests of an e-Learning subcommunity.

In the next sections we are going to detail the mechanisms RVL utilizes to accomplish the previously described functionality.

[2] For illustration purposes, the default namespaces are included in the example.

3. RVL: an RDF view definition language

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. Design choices for the RVL view definition language

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 sup ported by the view specification language?
(4) How can the output of view specifications be used in queries and other views?

In the following paragraphs, we will present the main design choices for RVL in order to address the aforementioned fundamental issues.

3.1.1. Logical data independence

Logical data independence is one of the most important properties that a view definition language should respect (recall the ANSI-SPARC three-level architecture [4]). It essentially requires that view specifications be independent of those of the source schemas and bases, while the semantics of existing virtual schemas be unaffected 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 to not only create *new* (meta)classes and properties (as in Fig. 2), but to also *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 those at the source. 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.

3.1.2. View instantiation capabilities

Besides the population of virtual (meta)classes and properties using, for instance, RQL queries (see Fig. 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.

3.1.3. Transformation expressiveness

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 [18,7]. Therefore, a view specification language should provide the ability to both create (for personalization purposes) and reconcile (for mediation purposes) quite different conceptual representations of the same application domain. For this reason, RVL is equipped with *expressive restructuring* capabilities 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" meta-classes 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) (see Table 1 in the Appendix).

3.1.4. 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 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 following paragraphs, 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 with respect to the RQL/RVL type system using the typing rules presented in Table 1 in the 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 from altering the type of one element using the instantiation operator, as we will subsequently see in this section.

In the following, we will sketch the functionality supported by each operator by using the more complex view illustrated in Fig. 3. This virtual schema is defined as a view on the schema of the motivating example in Fig. 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.

3.2.1. The instantiation operators "()"

The instantiation operator, denoted by "()", 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 a 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

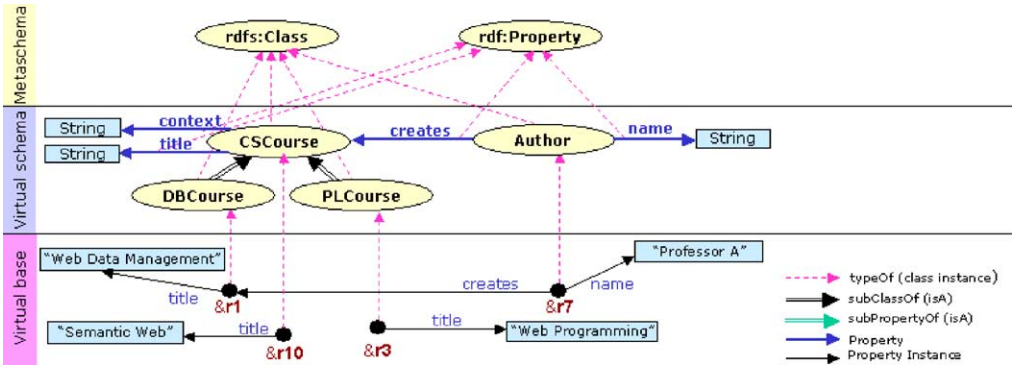


Fig. 3. A more complex RVL view *3.2.1 The instantiation operator* "()".

Fig. 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 in Table 1 respectively). In order to import a part (i.e., a *set*) of the classes defined in a source schema, we first need 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 with respect to 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), 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 highly diverse modeling 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 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 classes `Author` and `CSCourse`.

Due to the functional nature of RVL, the operands of the instantiation operator are not restricted to atoms (constants or variables), but can also be 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",range(ns1:createdBy),domain(ns1:createdBy));
```

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 Fig. 1, this RVL statement creates two instances of the metaclass *Property* with names `part Of` and has `Prerequisite` 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 Fig. 3 (see rules 11 and 12 in Table 1, 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 `CSCourses` published by himself, he/she can issue the following RVL statement through a suitable, easy-

to-use interface (presentation service), which automatically generates RQL/RVL statements (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 have already been defined in the view).

3.2.2. *The sub sumption operator "⟨ ⟩"*

The subsumption operator, denoted by "⟨ ⟩", 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<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 gives 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 traverses 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 bound to the corresponding (direct or 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 in a "connected" hierarchy related through subsumption subclasses (e.g., figures, exams, etc) to their ancestors(s) (e.g., *Learning_Object*). Since the use of appropriate labeling schemes for class (or property) Directed Acyclic Graphs (DAGs) [12] 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 the capturing of several modeling constructs recently proposed in OWL [13], such as inverse properties, synonyms of classes and proper-

ties or complex class definitions using boolean expressions and existential/universal quantifiers (supported by RQL filters) in a view.

3.3. The RQL/RVL internal logical framework

In order to fully capture the semantics of RQL queries and RVL views and to formally ground the ability to compose RVL views with RQL queries, we use an internal logical framework, which captures the semantics of RDF/S schemas and queries/views [11]. This first-order, relational framework is used to translate both queries and views into a common formalism providing methods to ensure the validity of logical operations. More specifically, this internal logical framework employs first-order relations together with some first-order constraints to model RDF/S and uses a signature with three sorts: Resource, Property and Class.[3] The relations used, along with their respective meaning, are listed below:

- C_EXT(*c*, *x*) iff the resource *x* is in the *proper extent* (i.e., it is a direct (proper) instance) of class *c*. According to the RDF specification, class extents can overlap due to multiple classification of resources;
- C_SUB(*c*, *d*) iff *c* is a (not necessarily direct) subclass of *d*;
- P_ROP(*c*, *p*, *d*) iff class *c* is the domain and class *d* is the range of property *p*;
- P_EXT(*x*, *p*, *y*) iff (*x*, *y*) is in the *proper extent* (i.e., it is a direct (proper)
- instance) of property *p*.[4] Like class extents, property extents may overlap;
- P_SUB(*p*, *q*) iff *p* is a (not necessarily direct) subproperty of *q*.

In order to be compliant with the RDF/S semantics, these relations must satisfy some *built-in* constraints imposed by the RQL/RVL data model. In particular, the domain and range of a property must be unique, while the subclass and subproperty relations must be reflexive, transitive and satisfy the following *subproperty/subclass compatibility* constraint:

$$\forall\, a,p,b,c,q,d\ \ \mathrm{P_SUB}(q,p) \wedge \mathrm{PROP}(a,p,b) \wedge \mathrm{PROP}(c,q,d) \longrightarrow \mathrm{C_SUB}(c,a) \wedge \mathrm{C_SUB}(d,b)$$

This means that if *q* is a subproperty of *p*, the domain and range of *q* are subclasses of the domain and range of *p*, respectively. Furthermore, we have the *property-class extent compatibility* constraint, i.e., any instance of a property *p* connects a pair of instances of some subclasses of the domain and range of *p*, respectively:

$$\forall\, a,p,b,x,y\ \ \mathrm{PROP}(a,p,b) \wedge \mathrm{P_EXT}(x,p,y) \longrightarrow \exists\, c,d\ \ \mathrm{C_SUB}(c,a) \wedge \mathrm{C_SUB}(d,b) \wedge \mathrm{C_EXT}(c,x) \wedge \mathrm{C_EXT}(d,y)$$

If we denote by Δ_{RDF} the set of dependencies (constraints) used to axiomatize the internal RDF/S model, then the following theorem holds:

Theorem 1. *It is decidable whether $\Delta_{RDF} \models d$ and whether $\Delta_{RDF} \models Q1 \subseteq Q2$, where d is an embedded implicational dependency, Q1, Q2 are conjunctive queries and $\subseteq$ is query containment.*

Using the relations C_SUB, PROP and P_SUB and the names of classes and properties defined in a schema as constants, we can straightforwardly translate the information embodied into an RDF/S schema to this internal framework as a set of relational facts (in Datalog parlance, an extensional database). For instance, some of the facts obtained from the schema of Fig. 1 are:

$$\mathrm{C_SUB}(\texttt{Course}, \texttt{Learning_Object})$$
$$\mathrm{PROP}(\texttt{Learning_Object}, \texttt{createdBy}, \texttt{Contributor})$$
$$\mathrm{P_SUB}(\texttt{partof}, \texttt{related})$$

Note that this set of facts will include all CSUB and PSUB instances due to reflexivity and will be "closed" under transitivity and under subproperty/subclass compatibility.

3.3.1. Translation of RQL queries into the internal logical framework

An RQL *conjunctive* query is a query of the form *ans(X)*: — *C1*, …, C_n where the C_i's are either RQL class or property patterns (as they appear in the RQL `FROM` clause) or equalities involving variables and/or constants and *X* is a tuple of variables or constants (range restrictions are also required). Many RQL

[3] For simplicity reasons, we ignore metaclasses and metaproperties in this discussion, but they can be handled easily in the same way.

[4] In our model, instances of properties are represented as ordered pairs of the resources they connect.

queries are in fact conjunctive queries, e.g., the query in section 2 can be written as:[5]

```
ans(Y, X, W) :- {Y;Course}createdBy{X}, {X}name{W},
         {Y}subject{Z}, Z = "Database Management"
```

Conjunctive RQL queries can then be translated into relational conjunctive queries in the internal logical framework. Indeed, according to the declarative semantics defined in [17], RQL patterns have the same meaning as conjunctions of relational atoms. For example:

RQL Pattern	*Internal Model Translation*
$\{X;\$C\}@P\{Y;\$D\}$	$\text{PROP}(a,p,b)$, $\text{P_SUB}(q,p)$, $\text{P_EXT}(x,q,y)$, $\text{C_SUB}(c,a)$, $\text{C_SUB}(d,b)$, $\text{C_EXT}(c,x)$, $\text{C_EXT}(d,y)$
$\{X;\$C\}@P\{Y\}$	$\text{PROP}(a,p,b)$, $\text{P_SUB}(q,p)$, $\text{P_EXT}(x,q,y)$, $\text{C_SUB}(c,a)$, $\text{C_EXT}(c,x)$
$\{X\}@P\{Y\}$	$\text{P_SUB}(q,p)$, $\text{P_EXT}(x,q,y)$

In the above RQL patterns, *X*, *Y* are resource variables, $*C*, $*D* are class variables (and can be replaced with constant class names), and *@P* is a property variable (that can also be replaced by a constant property name). Using these patterns, the RQL conjunctive query above translates internally to the following Datalog rule:

```
ans(y,x,w) :- PROP(a,createdBy,b),P_SUB(q1,createdBy),
              P_EXT(y,q1,x),C_SUB(Course,a),C_EXT(Course,y),
              P_SUB(q2,subject),P_EXT(y,q2,z),
              z="Database Management"
```

3.3.2. *Translation of RVL views into the logical framework*

In order to favor personalization, virtual RDF/S schemas can be also specified on top of the portal schema, as for instance the RVL schema shown in Fig. 3. If we restrict our attention to "conjunctive" RVL definitions, virtual class and property definitions and extents can also be written as rules, whose heads are as in the `VIEW` clause in RVL and bodies are just like conjunctive RQL queries. For instance:

[5] Namespaces are omitted for clarity reasons. Furthermore, for demonstration reasons, the "like" condition defined in the WHERE clause of the example query is replaced with equality.

```
Class(DBCourse) :-
Class(PLCourse) :-
Class(CSCourse) :-
CSCourse<DBCourse> :-
CSCourse<PLCourse> :-
Class(Author) :-
Property(creates, Author, CSCourse) :-
Property(name, Author,String) :-

DBCourse(Y):- {Y;Course}subject{Z}, Z="Database Management"
PLCourse(Y):- {Y;Course}subject{Z}, Z="Programming"
Author(X):-   {Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Database Management"
Author(X):-   {Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Programming"
creates(X,Y):-{Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Database Management"
creates(X,Y):-{Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Programming"
name(X,W):-   {Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Database Management",
              {X}name{W}
name(X,W):-   {Y;Course}createdBy{X}, {Y}subject{Z},
              Z="Programming", {X}name{W}
```

All the above rules can be translated into a set of conjunctive relational views. To accomplish this, the internal framework is equipped with similar relations as those presented in Section 3.3 in order to capture virtual classes and properties, as well as their virtual subsumption relationships, namely CLASSV, PROPV, CEXTV, PEXTV, CSUBV and PSUBV, respectively. Thus, for the rules presented previously, the conjunctive relational views created are:

```
CLASS_V(DBCourse) :-
CLASS_(PLCourse)  :-
CLASS_V(CSCourse) :-
C_SUB_V(DBCourse,CSCourse) :-
C_SUB_V(PLCourse,CSCourse) :-
CLASS_V(Author) :-
PROP_V(Author, creates, CSCourse) :-
PROP_V(Author, name, String) :-
C_EXT_V(DBCourse,y) :-  PROP(a,subject,b),P_SUB(q,subject),
                        P_EXT(y,q,z), C_SUB(Course,a),
                        C_EXT(Course,y),z="Database Management"
C_EXT_V(PLCourse,y) :-  PROP(a,subject,b),P_SUB(q,subject),
                        P_EXT(y,q,z), C_SUB(Course,a),
                        C_EXT(Course,y),z="Programming"
C_EXT_V(Author,x) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                        P_EXT(y,q1,x), C_SUB(Course,a1),
                        C_EXT(Course,y), P_SUB(q2,subject),
                        P_EXT(y,q2,z),z="Database Management"
C_EXT_V(Author,x) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                        P_EXT(y,q1,x), C_SUB(Course,a1),
                        C_EXT(Course,y), P_SUB(q2,subject),
                        P_EXT(y,q2,z),z="Programming"
P_EXT_V(x,creates,y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                        P_EXT(y,q1,x), C_SUB(Course,a1),
                        C_EXT(Course,y), P_SUB(q2,subject),
                        P_EXT(y,q2,z),z="Database Management"
P_EXT_V(x,creates,y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                        P_EXT(y,q1,x), C_SUB(Course,a1),
                        C_EXT(Course,y), P_SUB(q2,subject),
                        P_EXT(y,q2,z),z="Programming"
```

```
P_EXT(x,name,w) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                      P_EXT(y,q1,x), C_SUB(Course,a1),
                      C_EXT(Course,y), P_SUB(q2,subject),
                      P_EXT(y,q2,z),z="Database Management",
                      P_SUB(q3,name),P_EXT(x,q3,w)
P_EXT(x,name,w) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),
                      P_EXT(y,q1,x),C_SUB(Course,a1),
                      C_EXT(Course,y),P_SUB(q2,subject),
                      P_EXT(y,q2,z),z="Programming",
                      P_SUB(q3,name),P_EXT(x,q3,w)
```

Note that, the schema part of these conjunctive relational views needs to be "completed", i.e., closed, under reflexivity and transitivity.

3.3.3. Composing RQL queries with RVL views

Having set up a logical framework that captures the RDF/S semantics, we then translated both RQL queries and RVL views into this framework. The objective of this translation is to provide a logical framework into which queries formulated against a view are combined 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 [11]. Consider, for example, the following query, which retrieves the database courses created by the author named "Professor A":

```
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#
```

which translates to:

```
ans(Y) :- {X}creates{Y}, {X}name{Z}, Z="Professor A"
```

This rule is translated into the RQL/RVL internal model *of the view*, as follows:

```
ans(y) :- P_SUB_V(q',creates),P_EXT_V(x,q',y),
          P_SUB_V(q",name),P_EXT_V(x,q",z), z="Professor A"
```

Having translated both the query and the view definition into the internal logical framework, we can then compose this query with the view specification (all expressed in the internal models) by just performing a composition of (nonrecursive) Datalog programs. Note that P_SUB_*V*(q', *creates*) matches only the reflexivity instance PSUBV(creates, *creates*) (similarly for PSUB *V*(q'', *name*)). Thus, we obtain:

```
ans(y):- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
         C_SUB(Course,a1),C_EXT(Course,y),P_SUB(q2,subject),
         P_EXT(y,q2,z1),z1="Database Management",
         PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
         C_SUB(Course,c1),C_EXT(Course,v),P_SUB(r2,subject),
         P_EXT(v,r2,u),u="Database Management",P_SUB(r3,name),
         P_EXT(x,r3,z),z="Professor A"
ans(y):- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
         C_SUB(Course,a1),C_EXT(Course,y),P_SUB(q2,subject),
         P_EXT(y,q2,z1),z1="Database Management",
         PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
         C_SUB(Course,c1),C_EXT(Course,v),P_SUB(r2,subject),
         P_EXT(v,r2,u),u="Programming",P_SUB(r3,name),
         P_EXT(x,r3,z),z="Professor A"
ans(y):- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
         C_SUB(Course,a1),C_EXT(Course,y),P_SUB(q2,subject),
         P_EXT(y,q2,z1),z1="Programming",PROP(c1,createdBy,d1),
         P_SUB(r1,createdBy),P_EXT(v,r1,x),C_SUB(Course,c1),
         C_EXT(Course,v),P_SUB(r2,subject),P_EXT(v,r2,u),
         u="Database Management",P_SUB(r3,name),P_EXT(x,r3,z),
         z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
         C_SUB(Course,a1),C_EXT(Course,y),P_SUB(q2,subject),
         P_EXT(y,q2,z1),z1="Programming",PROP(c1,createdBy,d1),
         P_SUB(r1,createdBy),P_EXT(v,r1,x),C_SUB(Course,c1),
         C_EXT(Course,v),P_SUB(r2,subject),P_EXT(v,r2,u),
         u="Programming",P_SUB(r3,name),P_EXT(x,r3,z),
         z="Professor A"
```

As we can observe, the answer to the query posed on the view is the union of four queries, which we call—in order of listing—Q_1, Q_2, Q_3 and Q_4. Then, Q_1 and Q_4 minimize to Q' and Q'_4, respectively:

```
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Database Management",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Programming",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
```

However, Q'_1 is a subquery of (hence it contains) Q_3. Similarly, Q'_4 contains Q_3. Hence, the entire program can be minimized to $Q'_1 \cup Q'_4$! The result of translating this reformulated query back to RQL would be:

```
ans(Y) :- {Y;Course}createdBy{X},
          {Y}subject{Z1}, Z1="Database Management",
          {X}name{Z}, Z="Professor A"
ans(y) :- {Y;Course}createdBy{X},
          {Y}subject{Z1}, Z1="Programming",
          {X}name{Z}, Z="Professor A"
```

In conclusion, the query posed on the view is translated into a query posed to the source RDF/S description base, which can be actually evaluated (thus avoiding the computation of the view data in its entirety).

4. Related work

Several view specification languages have been proposed in the database literature. Most relevant to RVL is work conducted in the context of ODMG-compliant object-oriented DBMS, such as O_2 [1,22], MultiView [21], Chimera [16] and K2 [24]. 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.) and 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 [25], set-based operations have been introduced in order to define object-preserving views using an untyped version of RQL. Unlike the case of 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 [14], which relies on F-logic rules to define only virtual description bases. Unlike RVL also, 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, [3] 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.

Several issues need to be dealt with in order to fully support a view definition mechanism for RDF/S. An important issue is checking the *consistency* of view specifications, i.e., checking whether the graph they

produce satisfies the constraints of our model. We wish to develop methods for consistency checking that avoid the naive approach, in which the entire view data is constructed and then validated. Furthermore, 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.

Additionally, more issues remain open and require further investigation. Specifically, the cases dealt in this paper are conjunctive RQL queries and conjunctive RVL view definitions. In both these cases, we obtain a translation into non-recursive Datalog programs, to which we can apply well-known optimization techniques. We intend to study the conditions under which similar results can be obtained for a broader class of RQL queries and RVL view definitions. Another issue is the exploitation of existing knowledge about the source schemas and data in order to perform further optimizations during the reformulation process. The RQL/RVL internal model can also accommodate constraints, such as the ones expressible in OWL [13]. It will be interesting to study the optimization potential that stems from the use of such constraints (e.g., uniqueness or disjointness constraints) in query reformulation and minimization.

Appendix: RVL typing rules

The type system foreseen by RQL 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 create user-defined metaclasses of classes and properties respectively. The restrictions and inferences specified

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)
No 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)
	$e:\tau_{M_p}, e_1:\tau_1, \tau_1\in\{string, \tau_{M_c}, \tau_{M_p}, \tau_C, \tau_P\}$	
Property Creation	$\frac{e_2:\tau_2, \tau_2\in\{\tau_{M_c}, \tau_{M_p}, \tau_C\}, e_3:\tau_3, \tau_3\in\{\tau_{M_c}, \tau_{M_p}, \tau_C, \tau_L\}}{e(e_1,e_2,e_3):\tau_P[\tau_2,\tau_3]}$	(4)
MC subsumption	$\frac{e_1:\tau_{M_c}, e_2:\tau_{M_c}}{e_1\langle e_2\rangle:[\tau_{M_c}, \tau_{M_c}]}$	(5)
MP subsumption	$\frac{e_1:\tau_{M_p}, e_2:\tau_{M_p}}{e_1\langle e_2\rangle:[\tau_{M_p}, \tau_{M_p}]}$	(6)
Class subsumption	$\frac{e_1:\tau_C, e_2:\tau_C}{e_1\langle e_2\rangle:[\tau_C, \tau_C]}$	(7)
Property Subsumption	$\frac{e_1:\tau_P, e_2:\tau_P}{e_1\langle e_2\rangle:[\tau_P, \tau_P]}$	(8)
MC population	$\frac{e_1:\tau_{M_c}, e_2:\tau_C}{e_1\langle e_2\rangle:\tau_C}$	(9)
MP population	$\frac{e_1:\tau_{M_p}, e_2:\tau_P}{e_1\langle e_2\rangle:\tau_P}$	(10)
Class population	$\frac{e_1:\tau_C, e_2:\tau_U}{e_1\langle e_2\rangle:\tau_U}$	(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)

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*".

References

[1] S. Abiteboul, A. Bonner, Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, CO, 1991, pp. 238–247.

[2] Association for Computing Machinery, The ACM Computing Classification System, version: 1998, http://www.acm.org/class/1998/.

[3] Aidministrator Nederland bv: SeRQL user manual, version: 0.4 (2003), http://sesame.aidministrator.nl/publications/SeRQL.

[4] ANSI/X3/SPARC Study Group on Database Management Systems. Interim Report. ACM SIGMOD Bulletin 7 (1975) N2.

[5] T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, http://www.ietf.org/rfc/rfc2396.txt.

[6] T. Berners Lee, J. Hendler, O. Lassila: The Semantic Web. In: Scientific American (May, 2001), http://www.sciam.com/2001/0501issue/0501berners-lee.html.

[7] P. Bernstein, A. Levy, R. Pottinger, A Vision for Management of Complex Models, Microsoft Research Technical Report MSR-TR-2000-53 (2000), http://research.microsoft.com/scripts/pubs/view.asp?TRID=MSR-TR-2000-53.

[8] B.S. Bloom, D.R. Krathwohl (Eds.), Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook I, Cognitive Domain, Longman, New York, 1956.

[9] T. Bray, D. Hollander, A. Layman, Namespaces in XML, W3C Recomm. (1999).

[10] D. Brickley, R.V. Guha (Eds.), RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recomm., 10 February 2004.

[11] V. Christophides, G. Karvounarakis, I. Koffina, G. Kokkinidis, A. Magkanaraki, D. Plexousakis, G. Serfiotis, V. Tannen, The ICS-FORTH SWIM: a powerful semantic web integration middleware, in: Proceedings of the First International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universitat, Berlin, Germany, 2003, pp. 381–394.

[12] V. Christophides, D. Plexousakis, M. Scholl, S. Tourtounis, On labeling schemes for the semantic web, in: Proceedings of the 12th International World Wide Web Conference (WWW'03), Budapest, Hungary, 2003.

[13] M. Dean, G. Schreiber (Eds.), OWL Web Ontology Language Reference, W3C Recomm., 10 February 2004.

[14] S. Decker, M. Sintek, W. Nejdl, TRIPLE: A Logic for Reasoning with Parameterized Views over Semi-Structured Data, Tech. Rep., 2002, http://www.kbs.uni-hannover.de/Arbeiten/Publikationen/2002/tripleviews.pdf.

[15] D. Fensel, Y. Ding, E. Schulten, B. Omelayenko, G. Botquin, M. Brown, A. Flett, Product data integration in B2B e-commerce, IEEE Intell. Sys. 16 (4) (2001) 54–59.

[16] E. Bertino, G. Guerrini, B. Catania, J. Garcia-Molina, A formal model of views for object-oriented database systems, Theory Pract. Object Sys. 3 (3) (1997) 157–183.

[17] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, K. Tolle, RQL: a functional query language for RDF, in: P.M.D. Gray, L. Kerschberg, P.J.H. King, A. Poulovassilis (Eds.), The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data, Springer-Verlag, 2003, pp. 435–465.

[18] M. Klein, 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.

[19] O. Lassila, R. Swick, Resource description framework (RDF) model and syntax specification, W3C Recomm. (1999).

[20] A. Magkanaraki, G. Karvounarakis, T. Tuan Anh, V. Christophides, D. Plexousakis, Ontology Storage and Querying, Tech. Rep. 308, ICS-FORTH, Heraklion, Crete, Greece, 2002.

[21] E. Rundensteiner, MultiView: a methodology for supporting multiple view schemata in object-oriented databases, in: Proceedings of the 18th International Conference on Very Large Data Bases, VC, Canada, 1992, pp. 187–198.

[22] C. Souza dos Santos, S. Abiteboul, C. Delobel, Virtual schemas and bases, in: M. Jarke, J. Bubenko, K. Jeffery (Eds.), Proceedings of the Fourth International Conference on Extending Database Technology, St. John's College, Cambridge, UK, Lecture notes in Computer Science No. 779, 1994, pp. 81–94.

[23] M. Stratakis, V. Christophides, K. Keenoy, A. Magkanaraki, E-Learning Standards, SeLeNe Project Deliverable 2.1, 2003, http://www.dcs.bbk.ac.uk/ ap/projects/selene/.

[24] V. Tannen, S.b. Davidson, S. Harker, The information integration system K2, Elsevier, 2003.

[25] R. Volz, D. Oberle, R. Studer, Views for light-weight web ontologies, in: Proceedings of the ACM Symposium on Applied Computing SAC 2003, Melbourne, FL, USA, 2003.