Semantic Views for Controlled Access to the Semantic Web

Geetha Manjunath, Craig Sayers, Dave Reynolds, Venugopal KS, Swarup Kumar Mohalik, Badrinath R, John Ludd Recker, Malena Mesarina
HP Laboratories
HPL-2008-15
February 26, 2008*

semantic web, security, access control, semantics, views, RDF, OWL

The Semantic Web provides a good data integration and knowledge representation framework enabling sophisticated applications to reason and infer new information. Controlled access to data is extremely important when such data is being shared among multiple users. In this paper, we describe a framework that provides selective access to RDF data by supporting strict views over the semantic store. We propose a view specification language that enables specification of access restrictions using domain semantics. The view specification is materialized as an RDF graph model representing a sub ontology of the base ontology by our view system. Standard SPARQL queries and other graph operations can be performed on this view model. Our view mechanism has been implemented over the popular semantic web software framework, Jena and Joseki.

Internal Accession Date Only Approved for External Publication

Presented and published in Workshop on Semantic Web for Collaborative Knowledge Acquisition, SWeCKA'07

Semantic Views for Controlled Access to the Semantic Web

Geetha Manjunath, Craig Sayers, Dave Reynolds, Venugopal KS,
Swarup Kumar Mohalik[1], Badrinath R, John Ludd Recker, Malena Mesarina
Hewlett Packard Laboratories
Contact Email: geetha.manjunath@hp.com

Abstract

The Semantic Web provides a good data integration and knowledge representation framework enabling sophisticated applications to reason and infer new information. Controlled access to data is extremely important when such data is being shared among multiple users. In this paper, we describe a framework that provides selective access to RDF data by supporting strict views over the semantic store. We propose a view specification language that enables specification of access restrictions using domain semantics. The view specification is materialized as an RDF graph model representing a sub ontology of the base ontology by our view system. Standard SPARQL queries and other graph operations can be performed on this view model. Our view mechanism has been implemented over the popular semantic web software framework, Jena and Joseki.

1 Introduction

The Semantic Web [11] provides a common framework allowing data to be shared and reused across applications, enterprises, and communities. It is a collaborative effort led by the W3C with participation from a large number of researchers and industrial partners. Based on the Resource Description Framework (RDF) [13], the Semantic Web project intends to create a universal medium for exchange of data by providing semantics to information. It is particularly suited to model applications which involve distributed information problems such as integration of data from multiple sources, publication of shared vocabularies to enable interoperability, and development of resilient networks of systems which can cope with changes to the data models [1]. Restricted access to data in such an open framework is very important to ensure data privacy and to protect the system from malicious users.

Let us briefly look at the core elements of the Semantic Web framework. The principal technologies of the Semantic Web fit into a set of layered specifications. It is built over the foundation of URI's, XML, and XML namespaces. The first layer has the Resource Description Language, which is the primary data representation language specified as RDF Core. This flexible data representation does not require a pre-specified schema for structured data as it is based on a simple notion of RDF statements or triples consisting of subject, predicate and object for every assertion. The second layer is the Ontology Layer that includes the RDF Schema language [14] and the Web Ontology language (OWL) [16]. This enables specification of the domain semantics through concepts and known-relationships among them. Building on these core components is a standardized SQL-like query language called SPARQL [15] enabling RDF stores to be queried remotely.

We propose a view system that enables controlled access to the RDF store using an ontology-based specification. Views are an established technology for relational and object databases. Views are typically, used for enabling customized presentation of data as needed by an application or a user. They provide the programmer a degree of abstraction from the physical structure and schema of the underlying database. Analogous to the relational databases, there have been some efforts in providing views over semantic information as well. However, at a gross level the aim of most of this prior research has been for personalization, integration, and versioning of information (see section 6 for details). Our work focuses on supporting multiple views over an RDF store purely from a security standpoint – to provide authorized access to different elements of a semantic store.

The key contributions of this work are:
• A View Generation and Querying Framework that enables differential access to an RDF store – enabling user specific authorized access to semantic store.
• A View Mechanism that enables storing of the view specification in the same RDF store as the data; enabling administrative data, metadata and application data to be treated equally.
• A View Specification Language for Semantic Web which is expressed using the Web Ontology Language (OWL). We also define a View Ontology for View Specification.
• Implementation of the View Mechanism using the Jena Semantic Web Framework.

This paper is structured as follows. Section 2 provides a description of the abstract notion of views over semantic

[1] Swarup is now at General Motors India. This work was done when he was at Hewlett Packard.

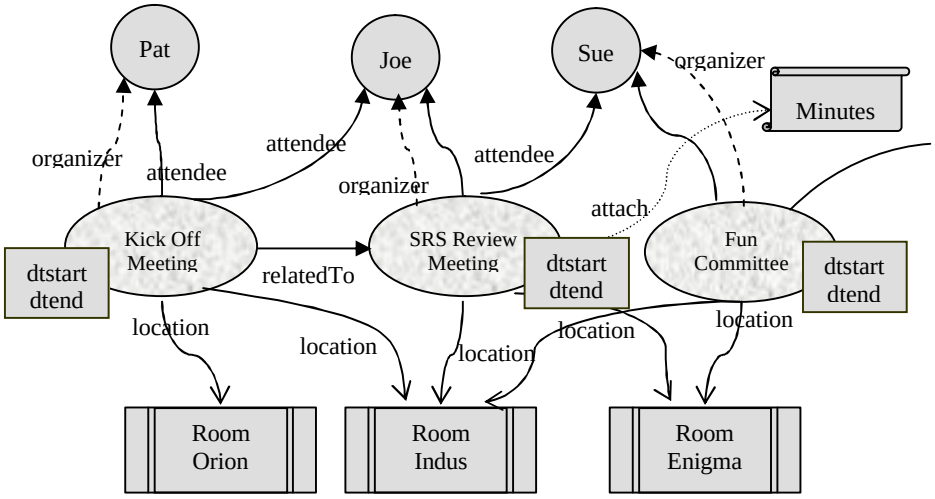

Figure 1: Example RDF graph for event scheduling

web with a simple example. In Section 3, we articulate the requirements of a view mechanism over the Semantic Web. The design of our framework is described in section 5 where we propose a view specification language and describe the different modules that processes it. More detailed implementation description is available in Section 6 which is followed by a section that narrates the related efforts for this problem of supporting views over semantic store.

2 Views

A view of a semantic store is a window to the base semantic store. Two primary uses of views are to enhance usability of data (providing personalized access to data that is relevant to an application) and to enable controlled access to data (giving access to part of the data depending upon the roles of its users). When used for access control, views provide a mechanism for enabling data security. As per the Semantic Web architecture, every ontology is associated with a set of RDF triples. A view defines a subset of these RDF triples of the base ontology to be visible. A typical use case would be a security administrator defining a view through a view specification and setting permissions to different users to access to different views.

Let us take an example. Consider the somewhat familiar activity of scheduling Video Conferences among selected invitees. Figure 1 has a sample RDF graph showing three Persons (Pat, Joe, Sue) invited to two conferencing events (meetings) scheduled in more than one room (among rooms Orion, Indus and Enigma). The graph represents a model in the iCalendar ontology [12] showing instances of the Vevent Class (representing the details of an event). Only a few properties and attributes of the Vevent class are shown here for simplicity, the complete event data would include other attributes of the attendees (email address), event schedule and so on.

Clearly, there will be a major privacy concern if a single global calendar data store containing all the details of the events scheduled in all the rooms, were visible to everyone. This is especially so if the video conferencing service is

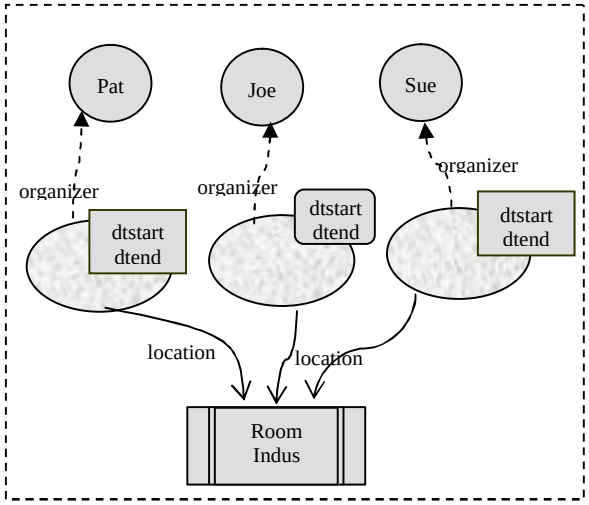

Figure 2: RDF graph seen by admin of Room Indus

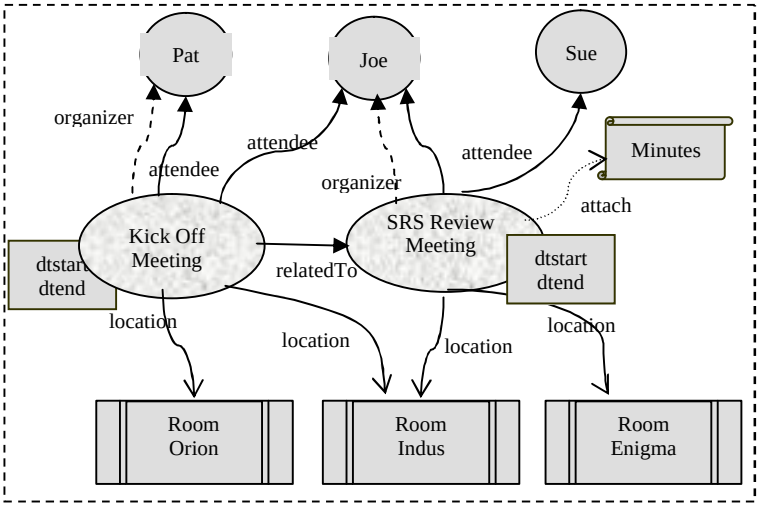

Figure 3: The RDF graph seen by Joe.

used by commercial organizations who do not want their competitors to know about their meetings and attendees. We would therefore like to restrict the view of the calendar data to a subset of the information that a person is allowed to see. For instance, the administrator of a conference room should be allowed to know when a particular room is needed and when it is free – so that he or she can schedule the maintenance activities in the free period and keep the room ready when needed. However, the attendee and event details are not needed for this function. So, administrators should see only scheduling details of all the events being held in the specific rooms that they administer. Similarly, the attendees of a particular event should be able to see all the details of the events but only for those that they are invited to.

Figure 2 and 3 show the different RDF graphs that should be seen by these two classes of users. The admin of Room Indus would only see the organizer name and time schedule of the events scheduled in Indus – all the remaining information is hidden. On the other hand, attendee Joe would see all the information about events – but only for those, he is invited for. As may be evident, simple removal of an attribute or a class is not sufficient to create these two nontrivial views. The two views need to be described using a semantic description. and our view mechanism intends to enable that. As we will see later, Figure 5 describes the view corresponding to Figure 2 using the proposed view specification language. We will use this simple calendaring example with the above two sample views in the rest of the document to explain the view mechanism.

3 Definitions

Let us now look at a more formal definition of views. An ontology is a data model that represents a knowledge domain and is used to reason about the objects in that domain and the relations between them. It conceptually represents a perceived world through concepts (Classes), attributes (properties, features, characteristics, or parameters that objects can have and share), and relationships (ways that objects can be related to one another).

The W3C standard for describing an ontology is using the Web Ontology Language (OWL). OWL contains a sequence of ontology elements, namely, annotations, axioms, and facts [18]. Class axioms and property axioms are used to define class and property identifiers, respectively. This definition could either be partial or complete specification of their characteristics. The axioms that define classes and properties can also be OWL restrictions that provide local constraints on properties of a class or OWL Class Descriptions that describe a class using Boolean combination of other classes. The second type ontology element, facts provide information about the individuals of a class. There can be two kinds of facts. The first kind states information about a particular individual, in the form of classes that the individual belongs to plus properties and values of that individual. The second kind of fact is used to state that two different resources identifiers refer to the same instance or different instances. Annotations can be used to record authorship and other information associated with the ontology.

On the other hand, to represent an ontology in Description Logics[2], a distinction is drawn between the so-called "TBox" (terminological box) and the "ABox" (assertional box). In general, the TBox contains sentences describing concepts and properties (property axioms, class axioms, property Restrictions and class Descriptions defined above) while the ABox contains what are called "ground" sentences stating where in the hierarchy individuals belong (facts that define relations between individuals and concepts).

In this paper, we borrow from Description Logic the above-simplified representation of an ontology for a formal description of views and use the OWL terminology to describe more practical aspects. We write an ontology O as (OT, OA) where OT represents the TBox of O and OA represents the ABox of O. Also for any pair (OT,OA) to qualify as an ontology, the assertions in the ABox OA must have terms (concepts, attributes and relationships) only from the TBox OT. In other words, OT represents the set of classes and properties, class axioms and property axioms while OA represents the assertions of the instance data that includes class membership, attribute values and relationships between two instances.

A view is basically, a sub-ontology of the base ontology with simple additional extensions to enable personalization of data. We define a view, V over a base ontology B as a terminological extension of a sub-ontology of B, where:

- An ontology V= (VT, VA) is a sub-ontology of the base ontology B = (BT , BA) if $VT \subseteq BT$ and $VA \subseteq BA$.
- A terminological extension of an ontology (T, A) is an ontology ($T \cup T'$, A) where T' introduces concepts with new names such that no new inferences are drawn in the extension regarding the base ontology. Specifically, if Statement S is valid in ($T \cup T'$, A) and conceptNames(S) $\subseteq$ T, then S is valid in (T, A) too. Here conceptNames(S) refers to classes referred in the statement.

In other words, T' should not include additional axioms over concepts and properties of T; definition of new terminology that enables additional classification of data for personalization is however possible.

The above definition of a view has two main benefits. Firstly, it ensures that a view is transparent to the application designed to interact with the base ontology. The concepts and property names that are valid in the base ontology are valid even through the view.

For example, if several instances of Vevent are masked in a view, the class extension of Vevent in the view is accessed just as we would access from the base ontology (though it is a subset of the base ontology). This is unlike other approaches where the subset of the class extension of base concept is made available as either a new concept name or the same concept name in a different namespace. This is an

[2] A family of knowledge representation languages that provide logic-based semantics to ontologies.

```
@prefix cal:
<http://www.w3.org/2002/12/cal/icaltzd#> .

cal:Vevent a owl:Class ;
      rdfs:comment "Provide a grouping of
properties that describe an event." .

cal:attendee a owl:ObjectProperty ;
      rdfs:domain cal:Vevent ;
      rdfs:range cal:Value_CAL-ADDRESS .
cal:organizer a owl:ObjectProperty ;
      rdfs:domain cal:Vevent ;
      rdfs:range cal:Value_CAL-ADDRESS .
cal:location a owl:DatatypeProperty .
cal:dtstart a owl:DatatypeProperty .
cal:dtend a owl:DatatypeProperty .
cal:relatedTo a owl:ObjectProperty ;
      rdfs:domain cal:Vevent ;
      rdfs:range cal:Vevent .
```

Table 1 : The TBox of the Base Ontology

important feature and requirement of our design since we want to be able to use views for providing access control – in a manner that can be transparent to the users. Queries from an application can now refer to base concept names in the namespace of the base ontology itself; while the result of the query may be different based on the view the application is allowed to see. Secondly, views can now be hierarchically constructed (views of views). This is a very powerful concept – enabling the view framework to limit all operations (such as queries, updates) to the RDF store only through defined views. The base ontology trivially, is also a view by this definition.

Let us now see the ABox and TBox of the sample calendaring example whose base ontology is the RDF

```
HP:Event1 rdf:type ical:Vevent
   ; ~~cal:attendee "Pat"~~
     ; ~~cal:attendee "Joe"~~
   : ~~rdfs:label "Kick Off Meeting"~~
   ; cal:organizer "Pat"
     ; ~~cal: location "Room Orion"~~
   ; cal:location "Room Indus"
     ;cal:dtstart "2006-04-25T04:30:00Z"
   ; cal:dtend  "2006-04-25T06:30:00Z"
   ~~; cal:relatedTo HP:Event2~~
   .
```

```
HP:Event2 rdf:type ical:Vevent
   ~~; cal:attendee "Joe"~~
     ; ~~cal:attendee "Sue"~~
   : ~~rdfs:label "SRS Review Meeting"~~
   ; cal:organizer "Joe"
     ; ~~cal: location "Room Enigma"~~
   ; cal:location "Room Indus"
     ; cal:dtstart "2006-05-20T04:30:00Z"
   ; cal:dtend  "2006-05-25T06:30:00Z"
   ~~; cal:relatedTo HP:Event1~~
   .
```

Table 2: The ABox corresponding to the two instances of Vevent of Figure 1. The elements that are stuck-off are not available in the view corresponding to Figure 2.

Calendar Ontology available at [12]. A part of that ontology that is relevant to the example under consideration[3] is given in Table 1. This forms the TBox of the Base Ontology. The ABox corresponding to two event instances of the RDF graph of Figure 1 is given in Table 2. The ABox of a view on this would only include a subset of the base assertions. The assertions that are not visible in a view corresponding to Figure 2 (the view of the administrator of Room Indus) are depicted with a strikethrough them in Table 2.

[3] The range of the attribute cal:relatedTo as per original specification is String Literal.

4 Requirements on Views

As mentioned earlier, our main focus of implementing the views framework was to enable access control to a common RDF store. In this section, we elaborate on the specific requirements of the views framework that we envisaged due to this.

- **This and Nothing Else**: One of the first requirements of the semantics of the views that we wanted is to limit the visible window to a subset of the RDF store to the specification. Even though personalized terminology (concept names) may be available to the user querying the view, the user should be stopped from performing a malicious query, using either the base or extended terminology. For example, the administrator of Room Indus should not be allowed to do a query to list the schedule of all events that have an attendee called Pat. Even though the output of the query, a subset of events happening in Indus, may seem like something that the user is allowed to see, the scope of the query includes attendee information which the room administrator does not have access to. The query should be restricted to a viewable model alone in order to enforce the required security – through explicit specification of what is/is not visible.
- **Multiple Simultaneous Views** on a base ontology: In order to support multiple user roles, we need to support existence of multiple simultaneous views with different view specifications – ensuring consistency of the views when the base ontology gets updated.
- **Queries through Views:** One of the primary operations using views would be queries. Since the user sees the base ontology through a view, the query expressions from the user would have the vocabulary from the base ontology and hence queries through views should support the same namespace and vocabulary of the base ontology, but restrict the query results as per the authorized view specification for the user.
- **View Specification in the RDF Store:** We want to store the view specification in RDF; enabling the administrator to perform queries on the view specifications in the same way as one would do for data.
- **Construction of complex views from other views**: We want to be able to compose new views through operations on earlier defined ones (merging of views,

intersection of views). We also want to be able to create views over views. For example, if B is the base ontology and V1 and V2 are views whose sub-ontology extraction rules are defined through a view specification language, we have V1 = ViewSpec1(B) and V2=ViewSpec2(B) . It should be possible to construct a new view V3 = foo(V1, V2) where foo is a graph operation[4] such as union, intersection. The framework should also enable a new view to be defined over another hierarchically, V4 = ViewSpec3(V1).

- **Updates through a View:** We would like to be able to allow updates through a view. Since each view is a sub-ontology of the base ontology, updates done through a view should be reflected in the base ontology as well (inverse of second point above).

5 The View Framework

The high-level architecture of our view mechanism is shown in Figure 4. Our approach is to create a new RDF model corresponding to every view and restrict query operations on the same. The view framework consists of a view generator which takes in a view specification and generates an appropriate View Model as a subset of the base model. A View Updater watches for updates to the base model to make the relevant view models consistent. In the rest of this section, we look at the details of the view specification language, the view generation and, the view update module.

5.1 View Specification Language:

We propose a simple view specification language to describe the concept names and properties of the base ontology that would be visible in a view along with a semantic description of the instance assertions that would be included in the view. We chose to use OWL as the syntax for the view specification language. This provides the following nice advantages:

- Inconsistencies in the view specification can, in many cases, be detected early by reasoning over the ontology
- Existing semantic web tools can be used for parsing and analyzing the view specification.
- The view specification itself can be stored along with application data and metadata in an RDF store. This enables queries on the view specification such as listing concept names that are viewable through a specific view.

Referring back to our calendaring example, the view that is visible to the administrator of Room Indus (Figure 2) is defined by the view specification of Figure 5. The user first defines an instance of the View class and selects the TBox and ABox elements that need to be included in the view independently, through SelectProp/SelectClass and SelectInd properties respectively. The view specification says that the Vevent class is visible with selected attributes only – namely, the organizer, start and end time of the event. To describe the allowed instances of the class Vevent, a new class called "IndusEvents" is defined and is made visible through SelectInd. It may be noted that the specification is intuitive to the user as it specifies the semantics of the filtering using terms from the TBox of the

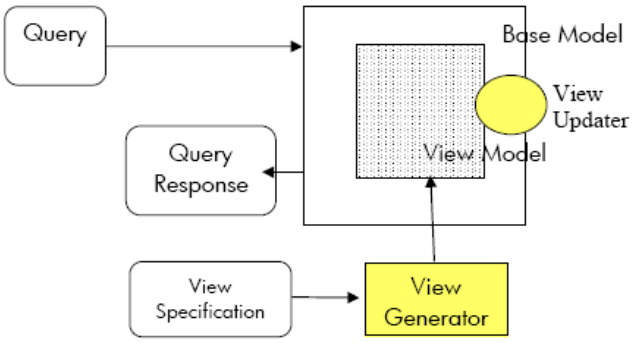

Figure 4: A high-level overview of our View Framework

Base Ontology. The view administrator needs to give this view specification only once and this would hold for all the future RDF data (in this case, additional instances of Vevent).

This specification language is our initial attempt towards representing access control restrictions on knowledge using semantics of the domain. A more detailed description of the view specification language is provided in the rest of the section.

Namespace and vocabulary

As a part of the proposed view language, we have defined a view Namespace and Vocabulary using which the user can provide a semantic definition for a specific view. Terminological extensions to the base ontology can be made by defining new concept names and properties in the view specification using standard OWL syntax.

We define a view namespace with a view ontology which

```
<view:View rdf:ID="RoomAdminView">
  <View:SelectClass rdf:about="cal:VEvent">
  <View:SelectProp
rdf:about="cal:organizer">
  <View:SelectProp rdf:about="cal:dtstart">
  <View:SelectProp rdf:about="cal:dtend">
  <View:SelectInd rdf:about="#IndusEvents">
</view:View>

<owl:Class rdf:Id="IndusEvents">
  <owl:intersectionOf rdf:type="Collection">
    <owl:Class rdf:about="cal:VEvent">
    <owl:Restriction>
      <owl:onProperty
rdf:resource="cal:location"/>
      <owl:hasValue
rdf:resource="#RoomIndus"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figure 5: The View Specification for the administrators of Room Indus

[4] The description of the complete semantics of the graph operations on view models is not in the scope of the current document.

defines a primary concept name called View at http://hpl.hp.com/research/sw/view. Every view specification would define an instance of this View class with properties specifying the visibility criterion for that view. These properties are mainly used to specify three kinds of information. First, to specify the classes and properties of the base model (TBox) that would be visible in the view. Second, specifying the semantics of selecting the instance data (ABox) and third: a new way of defining new classes using query strings, that would be used for selection of individuals. Table 3 at the end of this document gives a brief description of the semantics of the different keywords or Vocabulary of the view specification.

Positive and Negative views

To simplify the expression of semantics of the view, we categorize view specifications as either Positive or Negative, based on the structure and elements used in the specification. A positive view specification states the concept names, properties and instances that are visible, while a negative view specification states those that are not visible. Within a given set of ontologies, every positive view has an equivalent negative view and vice versa. Our view specification language supports both negative and positive assertions and includes keywords that have either positive or negative semantics (view:SelectClass and view:RemoveClass for example)

Use of negative specification when the end-user is allowed to access all but a few classes from the base ontology is likely to result in a shorter specification. In some cases, the view generation time may also be reduced due to the reduced number of RDF triples that need to be manipulated. More importantly, support for positive and negative views aids in creating view specifications that are in OWL-Lite instead of OWL-DL by eliminating the need for complementOf operator. While we believe that a language support for this dual mode of specification improves the ease of use of the view language, the proposed language has not yet been widely used for us to be convinced of this fact.

We employ a predefined prioritized handling of the view keywords in the view generation module to resolve conflicts and to avoid ambiguities in selection of instances. For both TBox and ABox selections, the positive directives (Select*) are first processed and then negative directives (Remove*) are honored. This is over and above the semantics mentioned in the Table 3. In the rest of this section, we look at more details of this view specification.

Specifying the TBox of a View :

As mentioned earlier, the user selects the TBox and ABox elements that need to be included in the view separately. To selectively include TBox elements into a view, we define a property called “Select” whose subject is the instance of the View class representing the view specification. The classes and properties of the base model that would be allowed in the view are specified as the objects of this Select property. Properties SelectClass and SelectProp are sub-properties of this class and can be used for better expressiveness. Analogous to these new properties for a positive specification, we have defined properties Remove, RemoveClass and RemoveProp to enable negative specification.

In our calendaring example, the room administrator is allowed to see only the class Vevent and a few properties of the event instances. The positive view specification for that, corresponding to Figure 2, would be:

```
<View:SelectClass rdf:about=”cal:Vevent” />
<View:SelectProp rdf:about=”cal:organizer” />
<View:SelectProp rdf:about=”cal:dtstart” />
<View:SelectProp rdf:about=”cal:dtend” />
```

Please note that the above specification is partial and defines only the visibility of the TBox. An equivalent negative specification (assuming a closed ontology) would be as follows:

```
<View:SelectAll />
<View:RemoveClass rdf:about=”#Minutes” />
<View:RemoveProp rdf:about=”cal:attendee” />
<View:RemoveProp rdf:about=”cal:label” />
<View:RemoveProp rdf:about=”cal:relatedTo” />
```

SelectAll includes all the concepts and properties of the base class into the view class and property definitions only, not instance data. We then list the properties of Vevent that cannot be viewed by the room administrator using the Remove directive. Please note that the negative specification may need to be changed every time there is an update to the TBox of the base ontology, since the default action for new concept names and properties is inclusion in the view, unlike the case in the positive specification.

The TBox selection of the view specification for the second view (Figure 3), where an attendee Joe would see all the classes and properties but is limited only to a subset of events (the instance data), would be just one line as there are no restrictions to the TBox.

```
<view:SelectAll/>
```

Specifying the ABox of the View:

Two properties in the view ontology are used for specifying the instance data to be included in the view specification semantically – SelectInd and SelectArc. SelectInd should be used to select individuals and SelectArc to select a subset of property arcs or edges. To select all individuals or resources of a particular class, the user specifies that class to be the object of property view:SelectInd. For providing restricted/partial access to instances of a class, user can define a new owl:Class describing the semantics for inclusion and specify that class to be an object of property view:SelectInd. Again, equivalent vocabularies for negative specification, namely RemoveInd, RemoveArc are also available in the language.

In the example view specification for a room administrator shown in Figure 5, we define a new class called IndusEvents and makes that to be of type view:SelectInd – in effect, populating the view with those event instances that belong

to the new class IndusEvents. For the attendee view of the calendaring example, where Joe would be able to see all the information about the events for which he was invited, the view specification is shown in Figure 6.

```
<view:View rdf:ID="JoesView">
  <view:SelectAll/>
  <view:SelectInd rdf:about="#JoeEvents" />
</view:View>

<owl:Class rdf:ID="JoeEvents">
  <owl:intersectionOf rdf:type="Collection">
    <owl:Class rdf:about="cal:VEvent">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#attendee"/>
      <owl:hasValue rdf:resource="#Joe"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figure 6: View Specification for Attendee Joe

Finally, as a special case, the complete base ontology can be made visible in a view using:

```
<view:View rdf:ID="Everything">
  <view:SelectAll />
  <view:SelectIndAll />
  <view:SelectArcAll />
</view:View>
```

Defining new classes

New classes may need to be defined in a view specification either for extending the view to include personalized vocabulary or for semantically expressing the selection of individuals or arcs (like class JoeEvents in Figure 6). There are two ways of defining new classes in our view specification. The method of using OWL syntax for defining new classes based on property restrictions was seen in Figure 5 and 6. Alternatively, we allow a new way of defining new classes through the use of a query string in our view specification. Using this mechanism, the class "JoeEvents" defined in the previous example can also be defined using a query as follows:

```
<view:DefineClass
  rdf:about="JoeEvents"
  view:query=
"SELECT ?x WHERE (?x cal:attendee "Joe") ;" />
```

This way of using queries to specify classes, basically provides an escape mechanism for some of the limitation of OWL syntax and increases the expressiveness of the view specification language. One such case would be need of numerical comparison; say, if we want to include all attendees, whose age is greater than 50.

Defining new classes using a query sometimes simplifies the view specification too. The new classes defined this way can be used in both positive and negative specifications. However, it may be noted that any kind of implicit relationships among these defined classes are not implicitly derived. For example, if two classes are related through a subsumption relationship (say one query has a conjunction of one more triple than the other), that will not be automatically inferred unless it is explicitly specified.

5.2 View Generation Module:

The view generation module reads in the view specification file and generates a view model on which further queries can be processed. There are three main steps in creating a view model from the base model. First, the view generator loads the view specification and creates an RDF model for the specification conforming to our View Ontology. Next, based on the view specification, the generator derives the concept names and properties (TBox) of the base ontology that are allowed in the view being defined. The visible concept names may include new ones introduced in the view specification (terminological extensions). Finally, the concept names in the view have to be populated with the allowed instance assertions (ABox) of the base model. The view generator analyses the semantic assertions in the view specification and populates the concepts accordingly.

As mentioned earlier, use of OWL for view specification greatly simplifies the parsing and analysis of the view specification. The view generator is in fact a semantic web application in itself! During loading of the view specification, the OWL loader detects syntax errors and inconsistencies in the specification too. Further, since the inclusion of instance assertions to a view is specified semantically (such as "make visible only events which use Room Indus"), the population of the classes with a subset of the instance assertions of the base model uses OWL inference again.

5.3 View Update

The View Update module ensures that the view mechanism maintains the view model consistent with the base model even while the base model gets modified. There are two modes in which the view generator can be invoked – the daemon mode and instance mode. In the daemon mode, the view generator materializes the view at its deployment time and serves all the queries directed to that view over this materialized view model. In the Instance Mode, every time a query on a view is made, the view model is materialized and query processed over it. As may be evident, when the view generator functions in the instance mode, the view model is consistent with the base model irrespective of the amount of changes and updates. While in the daemon mode, some additional effort is needed to ensure that the materialized view is consistent with the base model – and this is when the View Updater module comes in. The View Updater is event based; it watches for view changing events on the base model and updates the view model as needed.

There are fundamentally two kinds of changes that can occur in the base model; namely, changes in the schema (TBox) and those in the instance data (ABox). Typically, as

also in the application where we plan to deploy the view mechanism, we expect instance data updates to be more common than the schema changes. We plan to use a combination of the two extreme approaches mentioned above for implementing consistent views. We would materialize the view after every TBox/Schema change and use the event-based view updates for instance data changes. We also have controlled way of adding new data to the semantic store in our system. The View Updater is invoked with the new update information periodically to ensure consistency of the materialized view. This mechanism was further simplified in our target application as the system does not allow deletion of data and updates to instance data are only in the form of addition of new data. The updater, therefore, just peeks at the data being added and pulls in the inserted triples to relevant views only.

5.4 Query Rewriting

Let us briefly look at an alternate way of view generation that could potentially eliminate the need for view updates using one of the well-known ways of providing controlled access to any data - through filters. Filters for semantic store can be either on the input query or on the query response. Filtering just the query response alone would not ensure complete security of data; the domain of the search should also be restricted. Therefore, that leaves filtering or rewriting the input query as a feasible way of implementing queries on multiple views. Clearly, modifying input queries would limit the supported operation to query on views only. It would require separate corresponding mechanisms for other operations on the RDF model. It does also not ensure that the model being queried is a valid sub-ontology of the base model to be able to infer additional information within the view. On the positive side, such an implementation is similar to a view generator running in an instant mode and hence would not require a separate updater module. Also, in cases where the base model is huge and the only operation to be performed on the view model is query, then query rewriting is a good scalable approach.

6 Implementation and Results

We have implemented a preliminary version of the view mechanism described in this paper using the Jena Semantic Web Framework. Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, and includes a framework to include custom rule-based inference engine as well. Our implementation was on a Debian GNU/Linux Server deployed over Xen Virtual Machine on 2600-MHz AMD Opteron (TM) 252 Processor.

We support a two-phased approach to defining a view. First step is machine-guided view definition phase and second is the materialization of the view definition. We therefore have developed two Java modules using Jena-2.4; one for view generation and the other for machine-guided view specification.

The view generator has been interfaced to Jena through the Jena Assembler enabling use of the generated view model in any application that uses Jena assemblers. Using this, we have been able to successfully deploy multiple Joseki SPARQL services each serving different views of a base model. Since every such Joseki service would be accessible with unique service URL, standard web server authentication mechanisms can be used to control user access to the service. The visibility of selective data is made possible through our view mechanism – thus providing an end-to-end access control of RDF data.

We have developed a new Jena Model Assembler to generate view models. This enables any Jena application to create view models. In particular, to enable a query service on the view model, one could define a view model as a Joseki dataset. An example Joseki specification for providing a SPARQL service over a view model is given in Figure 7. A new model called ViewModel is defined with three properties viewSpec, origModel and viewName, which provide the model representing the view specification, the model representing the base model and the resource URI of the view instance, respectively. The dataset defined above can be used to define a Joseki service in a standard way and enables view generation on different types of base models – persistent, in-memory, inferred model and so on.

```
_:view1 rdf:type joseki:RDFDataSet ;
   rdfs:label "view1" ;
   joseki:defaultGraph
   [
   rdf:type _:ViewModel ;
   ja:viewSpec
        [ a ja:MemoryModel ;
        ja:content [ja:externalContent
        <file:Data/viewSpec1.owl> ] ;
        ] ;
   ja:origModel
        [ a ja:MemoryModel ;
        ja:content [ja:externalContent
        <file:Data/employee.n3> ] ;
        ] ;
        ]
   view:viewName "myView";
   .
```

Figure 7: Joseki Configuration Snippet describing a dataset of type ViewModel

We tested the View Generation with multiple ontologies namely employee database, temperature sensing, location sensing data and network topology data. We found that the proposed view language was very useful to extract a selective sub-graph of the RDF graph in all the cases. Our preliminary performance measurements showed In our current implementation, generating a view over a moderate dataset (50,000 triples) took 2 minutes. This was acceptable for our particular application that performs updates every 10 minutes, but is clearly too slow for many situations. Queries on that extracted model were also faster due to reduced number of triples that the queries targeted. We are exploring ways to improve the speed of view generation – such as view specification compilation and optimizing the

custom query made by the generator. In the rest of this section, we describe some of the internal details of our current view generation module and the machine guided view specification.

6.1 View Generation

The view generator (viewGen) takes in a view specification plus a reference to the data model for the Base Ontology and generates a view model. A single OWL file provides a single view definition. Additional attributes of the view such as whether it is a snapshot view or live view (with the periodicity of update) can also be specified at the time of view generation.

Top Level Algorithm

As mentioned in section 4, the view generator has three major steps: parsing of the view specification, extracting the TBox information and then populating the instance elements (Abox) into the view model. The detailed algorithm for view generation follows.

- Load Model using OWL FileManager
- Convert specification to simplified form (uses Jena Rule Engine)
- Include Select'ed classes and properties into the view Model
- c å view:Select, include triple t = <y rdf:type, c> , if t å baseModel
- Compute the membership of all class c å view:SelectInd using OWL inference
- Include above individuals into the View Model
- Compute the membership of all class r å view:RemoveInd using OWL inference
- Remove above individuals from the View Model

View Generation Using SPARQL query:

We use a variant of query rewriting in our view generation to materialize the view. Instead of rewriting every query that is intended towards the view, we convert every view specification to a SPARQL CONSTRUCT query on the base ontology. The result of this query would be an RDF graph representing the materialized view model. The view generator ensures that the view model is a valid ontology by identifying and inserting the missing triples to make it semantically complete. Every view specification is converted to an equivalent query that extracts eligible triples out of the base model for performing the user-given query on it. A part of the query that provides the semantics of "SelectInd" keyword is listed in Table 4. The keyword VIEW is replaced by resource URI of the selected view instance at selection time.

This approach of implementing the semantics of a language using a query is unique to our approach and enables us to easily introduce new keywords with well-defined semantics. In addition, the example query fragment given above is generic and hence would remain same for all view specifications. Alternately, we could convert every view specification to an equivalent SPARQL query that extracts the subset of allowed triples from base model into the view model (query rewriting).

In fact, this approach of generating a query for every view specification can be extended to work on relational databases as well – wherein a view specification is converted to a combination of multiple SQL queries on the relational database. This enables applications to function using a hybrid mode of data representation – relational database for its complete data and a RDF store of a subset of interesting data to perform reasoning on. The subset to be kept in RDF form is determined by the view specification language.

6.2 Machine Guided View Specification:

It is not always possible for the user to ensure that all the necessary concept names and properties are rightly included in a view. During the specification of the view definition, the tool viewSpec helps the user identify errors in the specification without the actual dataset of the base model being available. It also guides the user to include related concept names and properties in the view to make it a valid ontology. This module uses the Jena Rule Engine to determine dependencies between the concept names and properties of the Base Ontology and directs the user to "Select" new classes and properties as needed. It also determines the missing concept names and properties in a selection that are needed to make an ontological view. We define a rule set to infer the dependencies between concepts and properties of the base ontology, compute semantic completeness of the view model and use that to direct the user.

Semantic Completeness

Let us look a little more closely on the exact information that is imported from the base ontology for a view specification. As defined in an earlier section, an ontology

```
CONSTRUCT { ?x ?y ?z } WHERE
{{ VIEW view:SelectInd ?c . VIEW view:Select ?z .
?x rdf:type ?c . ?x rdf:type ?z . ?x ?y ?z} ,
UNION
{VIEW view:SelectInd ?c . VIEW view:Select ?y .
?x rdf:type ?c . ?y rdf:type owl:DataTypeProperty .
?x ?y ?z}
UNION
{VIEW view:SelectInd ?c . VIEW view:Select ?y .
?x rdf:type ?c . ?x ?y ?z . FILTER isLiteral(?z) }
UNION
{VIEW view:SelectInd ?c . VIEW view:Select ?y .
?x rdf:type ?c . ?x ?y ?z . FILTER isURI(?z) }
UNION
{VIEW view:SelectInd ?c . VIEW view:Select ?y .
?x rdf:type ?c . VIEW view:SelectInd ?c2 . ?z
rdf:type ?c2 . ?x ?y ?z}
}
```

Table 4: SPARQL query for SelectInd

consists of mainly three elements: (a) Concept names and property names (b) Descriptions of concepts and properties (Concept axioms and Role axioms) (c) Instance data, the individuals of the defined concepts (Facts).

We note that the view model created by the view generator would depend upon the type of information we import from the base model into the view model. The semantics mentioned in Table 3 and that implemented by us so far include only items (a) and (c) above. The role and concept axioms are not included by default. This limits the information seen through the view to the instance data that is populated – no additional data can be inferred in the view model.

In order to enable additional inference in the view model, the role axioms and property axioms from the base model need to be included in the view model as well – which we call semantically complete view model. In our view framework, we chose to selectively include the axiom based on user inputs. We compute the class and property dependencies during the first phase of the view generation and guide the user towards specifying a semantically complete view. As a special case, the role axioms and property axioms of all classes can be included in the view if the RDFS classes and properties are explicitly made visible in the view specification.

In our calendaring example, when the property cal:relatedTo (if defined as a symmetric property) is selected, the property axiom, 'relatedTo rdf:type owl:SymmetricProperty' stating that relatedTo is a symmetric property would not be available in a view model. So, addition of a property assertion of the form "JournalX relatedTo EventY" on the view model would not infer "EventY relatedTo JournalX" if this update to the view model is independent of the base model. However, if the property owl:symmetricProperty is selected in the specification, the role axiom is also part of the view model and hence additional data can be reasoned within the view model itself. On the contrary, if the update is propagated to the base model, the new triple will be inferred in the base model and made visible back through the view; this would be true whether or not the property axiom is itself visible via the view.

6.3 Storing view specification in RDF

As the view specification is expressed in RDF/OWL, we can store the view definition in the same RDF store as the base ontology model. The view generator creates an instance of the view Ontology for this new specification and stores it in the RDF store. A section of the View Ontology is given in Figure 8 above.

```
<owl:Class rdf:Id="View">
   <owl:DataTypeProperty rdf:Id="#Select"/>
   <owl:DataTypeProperty rdf:Id="#SelectInd"/>
   <owl:DataTypeProperty rdf:Id="#Remove"/>
   <owl:DataTypeProperty rdf:Id="#RemoveInd"/>
   <owl:DataTypeProperty rdf:Id="#SelectAll"/>
   <owl:ObjectProperty rdf:Id="#generationMode" >
         <rdfs:range rdf:Id="#viewParameters"/>
   </owl:ObjectProperty>
   <owl:DataTypeProperty rdf:Id="#viewName" />
</owl:Class>
```

Figure 8: View Ontology

7 Related Work

Views are an established technology for relational and object databases. For example, database views allow transformation of the database schema and combining of multiple database tables to create a new virtual table. Some view implementations on databases also facilitate interoperability among the different components using mediators and wrappers [21]. However, in a classical approach to views on databases, one cannot provide a conceptual description of views and hence the semantics of the views remain unclear. By contrast, our views over semantic information can be described using an ontology and hence can be more expressive and meaningful. We can use automated reasoning engines to perform analysis and can catch some user errors.

There have been some efforts in providing views over semantic web earlier, but the focus of most of the prior research has been personalization, integration, or versioning of information [3]. Our work differs from them by supporting multiple views over an RDF store purely from a security standpoint – to provide authorized access to different elements of a semantic store. In addition, since our view specification language is OWL, our framework supports semantic definition of views.

Raphael Volz et al describes a view language and a system (called KAON) [6, 7] that provides semantic classification of views by defining views over classes and views over properties. They derive query expressions from the specification to populate the new class or property. One of the key differences of our work with respect theirs is that the new classes (views over classes) they define need to have a name different than the base classes. While this allows the view to easily introduce new terminology, it removes the transparency of terminology desired for access control mechanism.

Maganaraki et al explain an approach to a view mechanism called the RVL lens [8] which is a declarative view definition language for virtual RDF description bases and schemas that allows data restructuring as well as hierarchical view construction. Our view definition language based on OWL also allows hierarchical view construction and enables specification of additional semantics. So, inconsistencies in the specification can be identified using OWL reasoner. It does not also involve learning new syntax and paradigm for the specification.

Pavan Reddivari et al propose a security mechanism for the Semantic Web that specifies restrictions on creation, modification and browsing of the RDF stores [19]. The

access control specifications in their framework are in RDF but are at triple level -requiring literally every triplet to be marked as either 'use' or 'see'. Our view specification language can be used to specify the restrictions in a more semantic form – using the vocabulary of the ontology alone. Sebastian Dietzold et al use SWRL (Semantic Web Rule Language) to specify the rules of visibility in their recent effort for access control on a triple store [20]. Similar to our approach, they generate a view model (that they call as a virtual model) from the base model using the rules specified by the user.

Various optimizations have been used in literature to extract an optimal subOntology from a base Ontology for a given specification [2, 4, 10]. These optimizations ensure that the resultant subOntology provides some nice properties such as requirement consistency, well-formedness, semantic completeness and is represented in the extreme simplified form. Their work provides a good theoretical basis for our work. We employ a similar mechanism when we guide the user towards a semantically complete view specification during the interactive view specification phase.

A fine grain role-based access control system has been used by Dave Banks et al in [22] for a personal knowledge base, ePerson Snippet Manager. The approach used by them is to define a set of roles, assign patterns (Query-By-Example patterns) that define what is visible for each role, and then assign roles to users. Role based access control is possible even with our system. We do allow query-based specification of the visibility of data like them; in addition, we support specification of visibility through ontology.

The work by Edward Hung et al [23] supports a means of extracting interesting triples from an RDF store by extends the RDQL query language to include GROUPBY and aggregate operators. We perform a similar operation during view generation using standard query syntax. They support an interesting subset of views by their approach.

8 Conclusion and Next Steps

In summary, security of data and meta data stored in an RDF store is important for real-life applications that use semantic web technologies to integrate multiple data sources and perform reasoning on them. We describe the design and implementation of a view mechanism that aids in enabling controlled access to an RDF store. We propose a view specification language based on the Web Ontology Language (OWL) which enables semantic description of the access restrictions, storing of the access controls in the RDF store, and construction of complex views using other views. The materialized view model is a sub-ontology of the base ontology and hence is amenable for further RDF operations. When our View Generation framework was tested with multiple ontologies, we found that the proposed view language was very useful to extract a selective sub-graph of the RDF graph. While performance was acceptable for our particular application, it may not be sufficient for other situations. We are exploring ways to improve the speed of view generation.

Our intention is to host different views of a base model as different Joseki services and direct user queries to the right service. In that regard, we are exploring use of named graphs to represent individual materialized views. We are also investigating use of other security techniques for enforcing authorized user access.

We are also working on enhancing our view specification language to include mechanisms that help in describing views over integrated data sources. As a specific feature of the view language, we are exploring view keywords that refer to namespaces and hence specify visibility of classes and properties from a namespace. We believe this would improve the expressbility of the view language. Further, we plan to include a template based view specification system during the view definition stage to simplify specification of similar views. Support for view generation through the Jena API is one of the other usability features we would like to add in future.

Acknowledgements

We are thankful to the Jena Development Team from HP Labs Bristol, for providing the Jena API which simplified the implementation of our framework. We are grateful to Kevin Wilkinson for reviewing our work and providing us the knowledge on the Jena Database Backend. We are thankful to our colleagues Nic Lyons and Tyler Close for their comments and feedback on this work.

References

1. *An assessment of RDF/OWL modeling*, Dave Reynolds, Carol Thompson, Jishnu Mukerji, Derek Coleman, Digital Media Systems Laboratory, HP Laboratories Bristol, HPL-2005-189, October 28, 2005
2. *Semantic Completeness in Sub-ontology Extraction Using Distributed Methods*, Mehul Bhatt, Carlo Wouters, Andrew Flahive, Wenny Rahayu, and David Taniar, ICCSA 2004, LNCS 3045, pp. 508–517, 2004. ©Springer-Verlag Berlin Heidelberg 2004
3. *Ontology versioning on the Semantic Web*, Michel Klein and Dieter Fensel, Vrije Universiteit Amsterdam
4. *A practical walkthrough of the ontology derivation rules*, Wouters, C., Dillon, T., Rahayu, W., et. al:. DEXA2002 (2002) 259-268
5. *A practical approach to the derivation of materialized ontology view*, Wouters, C., Dillon, T., Rahayu, W., et. al:. In: Web Information Systems. Idea Group Publishing (2004)
6. *Views for light-weight web ontologies*, Raphael Volz, Daniel Oberle, Rudi Studer, SAC2003
7. *Implementing Views for light-weight Web Ontologies*, Raphael Volz, Daniel Oberle, Rudi Studer, IDEAS2003

8. *Viewing the Semantic Web through RVL Lenses*, Aimilia Maganaraki, Val Tannen, Vassilis Christophides, Dimitris Plexousakis, ISWC03
9. *XML Views*, Rajagopal Rajugan1, Elizabeth Chang, Tharam S Dillon, and Ling Feng
10. *SEKT, Semantically Enabled Knowledge Technologies*, http://www.sektproject.com
11. *Semantic Web*, http://www.w3.org/2001/sw/, www.semanticweb.org/
12. *iCalendar RDF/XML Schema, the OWL ontology corresponding to RFC 2445* http://www.w3.org/2002/12/cal/icaltzd
13. *Resource Description Framework*, RDF, http://www.w3.org/RDF
14. *RDFS, Resource Description Framework, (RDF) Schema Specification 1.0*, W3C Candidate Recommendation 27 March 2000, http://www.w3.org/TR/2000/CR-rdfschema-20000327/
15. *SPARQL Query Language for RDF*, http://www.w3.org/TR/rdf-sparql-query/
16. *OWL Web Ontology Language Overview*, http://www.w3.org/TR/owl-features/
17. *Web-calculus*, http://www.waterken.com/dev/Web/
18. *OWL Web Ontology Language, Semantics and Abstract Syntax*, http://www.w3.org/TR/owl-semantics/
19. *Policy based access control for an RDF store*, Pavan Reddivari, Tim Finin, Anupam Joshi, University of Maryland.
20. *Access Control on RDF Triple Stores from a Semantik Wiki Perspective*, Sebastian Dietzold and S¨oren Auer, University of Pennsylvania. 2nd Workshop on Scripting for the Semantic Web, June 2006
21. *Intelligent Integration of Information*, Wiederhold Kluwer Academic Publishers, 1993
22. *The ePerson Snippet Manager: a Semantic Web Application*, Dave Banks, Steve Cayzer, Ian Dickinson, Dave Reynolds, HP Laboratories Bristol, HPL-2002-328, November 2002
23. *RDF Aggregate Queries and Views*, Edward Hung, Yu Deng, V.S. Subrahmanian, ICDE 2005
24. *Implementing Views for Controlled Access to the Semantic Web*, Geetha Manjunath, Craig Sayers, Dave Reynolds, Venugopal KS, Swarup Kumar Mohalik, John Ludd Recker, Malena Mesarina, International Workshop on Semantic Web for Collaborative Knowledge Management, SWeCKa, Jan 2007

TBox Selection

- <view:Select className> or <view:SelectClass className>
  - Marks the class to be visible in the View
- <view:Select propertyName> or view:SelectProp propertyName>
  - Marks the property to be visible in the view
  - Domain and Range Classes should be selected separately
- <view:SelectAll>
  - Marks all properties and classes to be visible
  - This would be useful when the view restriction is only for Abox elements or when used in conjunction with view:Remove
- <view:Remove className> or <view:RemoveClass className>
  - Removes class from view and all arcs to and from it
- <view:Remove propertyName> or <view:RemoveProp propertyName>
  - Removes all the arcs with propertyName from view

ABox Selection

- <view:SelectInd className>
  - Includes individuals/resources of type className
  - Adds membership arcs from the above to marked classes if applicable.
  - Adds marked data type property arcs from selected individuals
  - Adds marked object type property arcs among selected individuals
- <view:SelectIndAll>
  - Includes all individuals of marked classes with the same semantics as SelectInd per class
- <view:SelectArc propName subjClass objClass>
  - All triples <s, p, o> are added if
    - <s, p, o> å Base Ontology
    - s å subjClass and s å a marked class
    - o å objClass and o å a marked class
    - p is a marked property
  - If subjClass and objClass are not specified, all triples <s, p, o> from the base class are added for all selected individuals s and p.
- <view:SelectArcAll>
  - Includes all arcs for marked Properties to and from selected individuals (default behavior)
- <view:RemoveInd className>
  - Removes the individuals of type className
  - Removes all arcs to or from the above individuals
- <view:RemoveArc propName subjClass objClass>
  - Removes all triples <s, p, o> iff
    - s å subjClass
    - o å objClass
    - p = propName
- <view:defineClass className queryString>
  - A new owl class consisting of the resources from the query response is created
    - Query should have a single unbound variable
    - This class can be further used in Abox selections

Table 3 : Semantics of our View Vocabulary