Journal of Web Engineering, Vol. 6, No. 4 (2007) 337–359
© Rinton Press

VQL – THE QUERY LANGUAGE FOR SEMANTICALLY INFORMATION RETRIEVAL IN THE SemanticLIFE DIGITAL MEMORY FRAMEWORK

HANH HUU HOANG[1,2], A MIN TJOA[1]
[1] *Institute of Software Technology and Interactive Systems*
Vienna University of Technology
Favoritenstrasse 9-11/188, A-1040 Vienna, Austria
{hanh, amin}@ifs.tuwien.ac.at

[2] *Department of Information Technology*
College of Sciences - Hue University
77 Nguyen Hue Street, Hue City, Vietnam
hanh@hueuni.edu.vn

Received June 01, 2007
Revised December 01, 2007

This paper presents an approach of supporting users in formulating queries in the Semantic Web applications. We create an innovative mediator environment containing a lighter query language to help the users in the challenging task of making unambiguous requests. The query language is designed firstly for the SemanticLIFE's Virtual Query System [1] and could be used for other Semantic Web applications. The proposed query language, namely Virtual Query Language (VQL), is an effort of reducing complexity in the query formulation, especially from the user-side. Furthermore, it simplifies the communication between components of the SemanticLIFE system with the VQS, and increases the portability of the system.

Keywords: Semantic Web, Semantic query language, Ontological information retrieval

1 Motivation

Making unambiguous queries in the Semantic Web applications is a challenging task for users. The SemanticLIFE[2]'s Virtual Query System (VQS) is an attempt to overcome this challenge. The VQS is a *front-end* approach for user-oriented information retrieval [1].

The issue is on the user-side where users are required to formulate queries for their information of interest. The current query languages for RDF[a], DAML[b] and more generally for semi-structured data provide very expressive mechanisms which are suitable for the back-end querying mechanism. To users, who are inexperienced with them, these query languages are too complicated to understand and use. Additionally, the communication of components of the SemanticLIFE system with the VQS requires the facility to transfer their requests without knowledge of any RDF query languages. Furthermore, there is an issue of the system portability, i.e. if the system is bound to specific RDF query language, we could have problems when shifting to another one.

[a]Resource Description Framework, http://www.w3.org/RDF/.
[b]DARPA Agent Markup Language, http://www.daml.org/.

In the effort of over coming the above challenges, we, in this paper, present a new query language namely *Virtual Query Language* (VQL). The language is used by the VQS for user query formulation and information retrieval in the SemanticLIFE system. The VQL is a much lighter weight language than RDF query languages; but it offers interesting features to complete the tasks of information querying in the Semantic Web applications.

The rest of this paper is organized as follows: firstly, similar approaches are briefly presented in Section 2. An introduction to the SemanticLIFE framework and the VQS is presented in Section 3. Next, details of the VQL are pointed out in Section 4. Section 5 introduces VQL query operators; and the issues of mapping a VQL query to the respective RDF query are then described in Section 6. The VQL context-aware query results presentation is presented in Section 7. Finally, the paper is concluded with an outlook of the future work.

2 Related Work

There are two main approaches to reduce the difficulty in generating queries from user-side in Semantic Web applications. The first trend is aiming at designing the friendly and interactive query user interfaces to guide users to build the queries. The high-profiled examples for this trend are GRQL [3] and SEWASIE [4].

GRQL - Graphical RQL - relies on the full power of the RDF/S data model for constructing on the fly queries expressed in RQL [5]. More precisely, a user can navigate graphically through the individual RDF/S class and property definitions and generate transparently the RQL path expressions required to access the resources of interest. These expressions capture accurately the meaning of its navigation steps through the class (or property) subsumption and/or associations. Additionally, users can enrich the generated queries with filtering conditions on the attributes of the currently visited class while they can easily specify the resource's class(es) appearing in the query result.

Another graphical query generation interface, SEWASIE, is described in [4]. Here, the user is given some domain-specific patterns to choose from as a starting point, which can be extended and customized by the user. The refinements to the query can either be additional property constraints to the classes or a replacement of another compatible class in the pattern such as a sub or superclass. This is performed through a clickable graphic visualization of the ontology neighborhood of the currently selected class.

The second approach of reducing complexity is the effort of creating much lighter query languages than expressive RDF query languages. Following this trend, the approach in [6] and another one known as GetData Query interface [7] are high-rate examples.

[6] describes a simple expressive constraint language for Semantic Web applications. At the core of this framework is a well-established semantic data model with an associated expressive constraint language. The framework defines a 'Constraint Interchange Format' in the form of RDF for the language, allowing each constraint to be defined as a resource in its own right.

Meanwhile, the approach of GetData Query interface of TAP expresses [7] the need of a much lighter weight interface for constructing complex queries. The reason is that the current query languages for RDF, DAML, and more generally for semi-structured data provide very expressive mechanisms that are aimed at making it easy to express complex queries. The idea of GetData is to design a simple query interface which enables to network accessible data presented as directed labeled graph. This approach provides a system which is very easy to build, support both type of users, data providers and data consumers.

Continuing this trend, we present in this article an effective and lighter weight query language—the VQL. This language is used for user query formulation and information querying in the SemanticLIFE system. The VQL offers interesting features to complete the tasks of information retrieval in the Semantic Web applications. In addition, the VQL is designed to assist the users to formulate queries in a simple manner and simplify the communication between components of the SemanticLIFE system with the query module—the VQS.

3 SemanticLIFE and Virtual Query System

3.1 The SemanticLIFE Framework

The SemanticLIFE framework is developed on a highly modular architecture to store, manage and retrieve the lifetime's information entities of individuals. It enables the acquisition and storage of data while giving annotations to email messages, browsed Web pages, phone calls, images, contacts, life events and other resources. It also provides intuitive and effective search mechanism based upon the stored semantics, and the semantically enriched user interfaces according to the user's needs. The ultimate goal of the project is to build a Personal Information Management (PIM) system over a human lifetime using ontologies as a basis for the representation of its content.

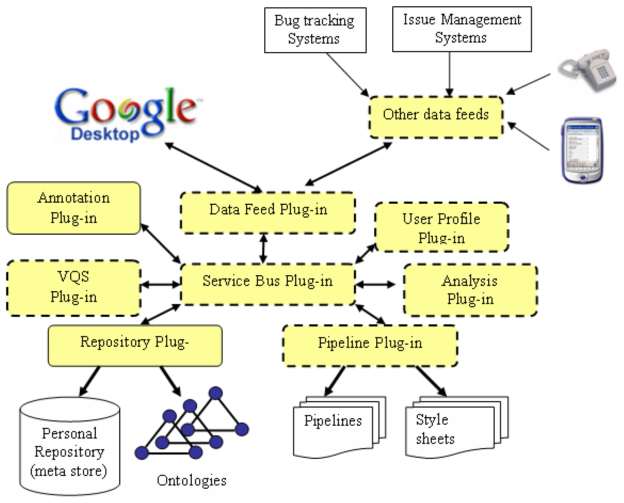


Fig. 1. The Architecture of the SemanticLIFE Framework.

The whole SemanticLIFE system has been designed as a set of interactive plug-ins that fit into the main application and this guarantees flexibility and extensibility of SemanticLIFE platform. Communication within the system is based on a service-oriented design with its loosely coupled components. To compose complex solutions and scenarios from atomic services which are offered by SemanticLIFE plug-ins, the Service Oriented Pipeline Architecture (SOPA) [8] has been introduced. SOPA provides a paradigm to describe the system-wide service compositions and also external Web services as pipelines.

The SemanticLIFE's system architecture overview is depicted in Fig. 1. Data with user annotation is fed into the system using a number of dedicated plug-ins from variety of data sources such as Google Desktop's[c] captured data, communication logs, and other application's metadata. The data objects are transferred to the analysis plug-in which contains a number of specific plug-ins which provide the semantic mark-up by applying a bunch of feature extraction methods and indexing techniques in a cascaded manner. The semi-structured and semantically enriched information objects are then ontologically stored via the repository plug-in in forms of RDF triples with their ontologies. This repository is called a *metastore*.

3.2 The Virtual Query System

3.2.1 Motivation and Architecture

Formulating non-ambiguous queries is always a demanding task to users as they are not aware of the semantics of the stored information. The goal of the Virtual Query System (VQS) is to overcome this problem by providing an ontology-based *virtual information* view of the data available in the system. If the user can *be aware of* what is inside of the system he/she can clearly and more precisely specify the queries against the data stored in the metastore.

The VQS system is primarily based-on the reduction of semantic ambiguities of the user query specifications at the very early stage of the retrieving process. The most important point in the VQS approach is that it provides *an image* of the real data sources of the system to the user. The user is aware of the data stored in the system when he/she generates the queries. As a result, the ambiguities in his/her requests will be much reduced.

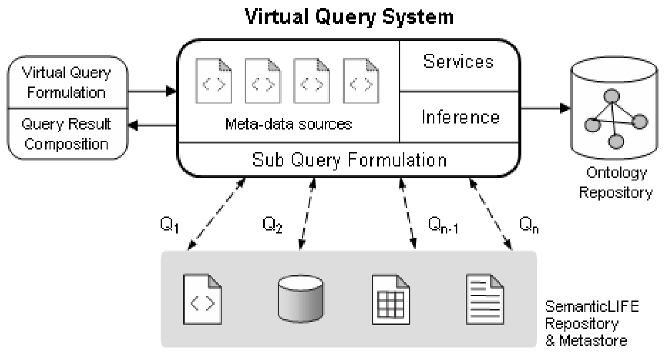


Fig. 2. The component architecture of the Virtual Query System.

The VQS's architecture is presented in Fig. 2. The *image* of the real data sources is reflected in the "Meta-data Sources" component—so-called the Virtual Data Component (VDC). The user, with the assistance of the Virtual Query Formulation unit, formulates the queries based on a *context ontology* in the VDC. The RDF queries will generated from the user's queries afterward at the Sub-Query Formulation unit using Services and the reasoning engine. The results will be transformed and aggregated before delivering to the user. Furthermore, the VQS could recommend the more relevant results to the user dependent on user's experiences, user ontology reflection or user profile [1].

[c]Google Desktop, http://desktop.google.com/.

3.2.2 VDC – The Virtual Data Component

The VDC, depicted as "Meta-data Sources" part in Fig. 2, contains the metadata of storage sources. This VQS crucial module acts as a virtual information layer to be delivered to the user. It enables the user to be aware of the semantics of the data sources stored and to specify more precise queries. The VQS collects metadata from data sources in the SemanticLIFE's metatastore. An analysis process is carried out on these metadata sources to get the semantic information. Then the processed information—so-called the *virtual information*—is stored in this module as a *context ontology*. Furthermore, this part is also referred as an image of the system database in further query processing, so-called the *context-aware querying* [9].

Based on the context ontology, the VQS refines the user's queries and creates the "real" RDF subqueries against the data sources in the metastore. In addition according the user's context, a set of predefined query templates over the virtual information is available for the user to select or customize. It would significantly support user in generating queries [10]. The metadata in the context ontology is still associated to the 'real' metadata in the system ontology. The metadata in the new context is used for the VQS's purpose in rendering information for presentation and reasoning during the querying process [10].

3.2.3 VQL – The Virtual Query Language

The VQL is specially designed for the VQS in the context of the SemanticLIFE framework to assist users and internal components interact with the system and each other without the RDF knowledge. Moreover, we intend to develop it to become a mediate semantic language that translate the XML-based queries to specific RDF queries that could be applied in SemanticLIFE-like PIM systems without caring RDF querying issues.

Initial VQL queries are transferred to RDF sub-queries for retrieving information from the metastore which is semantically organized in RDF triples and OWL[d] ontologies [2].

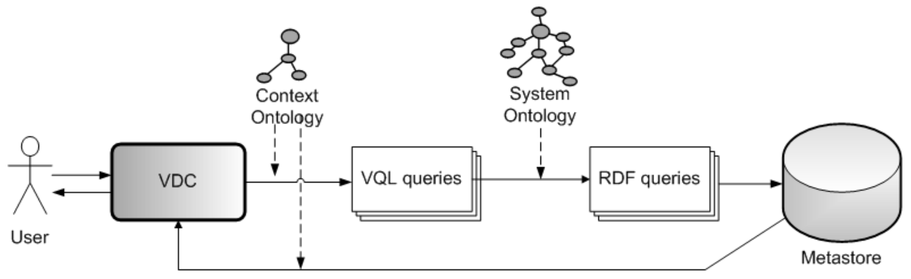


Fig. 3. RDF queries are generated from VQL queries with supports of ontologies.

As depicted in Fig. 3, with the support of the VDC the user can generate the virtual query over the virtual data. The VQS then analyzes the virtual query based on the context ontology. As the result, RDF subqueries from the initial query are generated for the specific underlined data sources. Finally, the results of subqueries are aggregated and represented to the user with regard to the context ontology.

[d]Web Ontology Language, http://www.w3.org/2004/OWL/ and http://www.w3.org/2007/OWL/

4 The Virtual Query Language

4.1 VQL's Objectives

A number query languages have been developed for the Semantic Web data such as RQL [5], RDQL [11], SPARQL [12], and iTQL [13]. The question is why do we need yet another query language?

All these query languages provide very expressive mechanisms that are aimed at making it easy to express complex queries. Unfortunately, with such expressive query languages, it is not easy for average users to construct queries, as well as to ask abstract information. What we need is a much lighter weight query language that is easier to use. A simple lightweight query system would be complementary to more complete query languages mentioned above. VQL is intended to be a simple query language with a support in a "semantic" manner for users' queries. In the context of the VQS and the SemanticLIFE system, we can summarize the objectives of the VQL as follows:

- VQL supports clients to query without knowledge of RDF query languages. The user just gives basic parameters needed information for VQL queries, and would receive the expected results;
- VQL assists users in navigating the system via semantic links or associations provided by powerful ontology-based operators;
- VQL simplifies the communication between the Query module and other parts, such that the components asking for information do not need to issues the RDF query statement, which is not easy for them. This feature also keeps the SemanticLIFE's components more independent;
- VQL enables the portability of the system. Actually, the SemanticLIFE and VQS choose a specific RDF query language for its back-end database. However, in the future, they probably could be shifted to another query language, so that this change does not effect other parts of the system, especially the interface with the system database.

4.2 The Syntax of VQL

4.2.1 Query Document Syntax

Definition 1 *A* VQL query document *has four parts: parameters, data sources, constraints (relations), and query results format.*

The syntax of a VQL query document is described in the query document schema depicted in Fig. 4. The first part contains *parameters* for specifying the information of interest. A *parameter* consists of a variable name, the criteria value, and additional attributes for sorting or eliminating unneeded information from the results.

The second part—*sources*—is used for specifying the sources where the information will have the extract from. Obviously, the information that need to be defined in the first part must be related to the sources specified in this part.

Thirdly, the *constraints* of the query—*relations*—are defined in the third part of the document. Here the relations between sources, parameters are combined using the VQL operators. Finally, the format of query results is identified in the fourth part. VQL supports

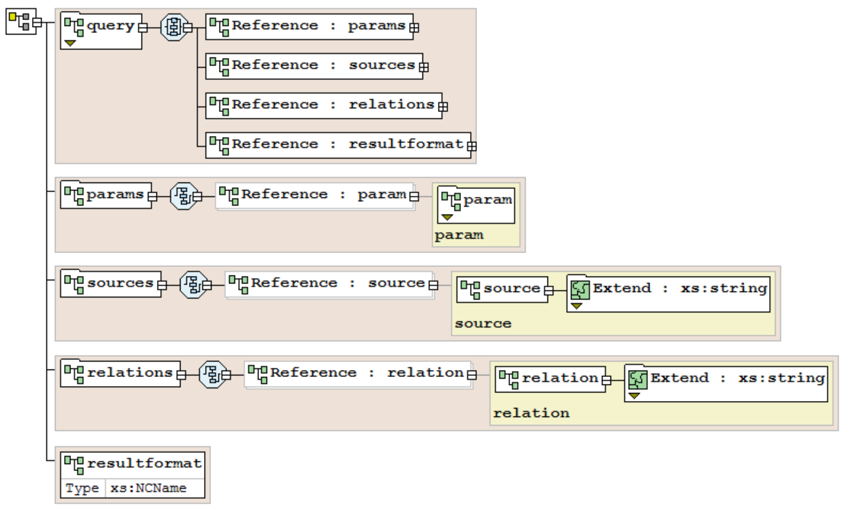


Fig. 4. The schema for general VQL queries.

four query results formats that are XML, text, RDF graph, and serialized objects of query result sets. This provides flexibility for clients to process the query results.

4.2.2 XML-based Format

A standard format for information exchange, an easy-to-use and familiar-to-clients format, a widely accepted standard, and a flexible and open format are the requirements for the VQL query document. We have considered some alternatives and decided to choose XML as the format for VQL query syntax. A XML-based VQL query is structured as follows:

```
<query type="data">
 <params>
  <param show="1" name="s1:messageTimeStamp">2005-11-01</param>
  <param show="0" name="s2:messageTimeStamp">2005-11-31</param>
 </params>
 <sources>
  <source name="fileupload">FileUpload</source>
  <source name="browsingsession">BrowsingSession</source>
 </sources>
 <relations>
  <relation id="1" param="s1" source="">dt:gt</relation>
  <relation id="2" param="s2" source="">dt:lt</relation>
 </relations>
 <resultformat>xml</resultformat>
</query>
```

Fig. 5. An Example of an XML-based VQL query.

- The top level or the body of query is the `<query>` element. Here, the type of the query must be specified in the `type` attribute. The reserved terms used for this attribute are `"data"`, `"schema"`, `"rdf1"` and `"rdf2"` for different VQL query types (see Section 4.4). For example, `<query type="data">` is a VQL data query.

- The second level contains required sub-elements. Depending on the type of a VQL query the elements are used respectively: for the data query, elements of `<params>`, `<sources>`, and `<relations>` are used once for each. These elements have their children specified in the third level. Fig. 5 is an example. For the schema query or the RDF type 1 query, we use only one `<statement>` element which contains an RDF query statement as depicted in Fig. 6. For the RDF type 2 query, elements of `<select>`, `<from>`, `<where>`, `<orderby>`, `<limit>` and `<offset>` are used in the same way, where last three are optional as described in Fig. 7. Moreover at this level, we must specify the query results format in the `<resultformat>` element by a reserved term such as `"xml"`, `"text"`, `"jason"`, `"rdf"`, or `"object"`.

- The third level elements are only applied for the VQL data queries. The elements are children of `<params>`, `<sources>`, and `<relations>`; and the tags consequently are `<param>`, `<source>`, and `<relation>` with their own attributes.

- `<param>` *element*: each parameter is identified by this element with required attributes: `show` and `name`. While `show` is set to 1 or 0 that means the result of this parameter is shown in the result sets or not; `name` has two parts: a *variable* and the $meta - information$ which are put together with `":"`, i.e. `variable:metainfo`. Besides, this element has two optional attributes known as `order="1"/"0"` and `exclude="string"`. The `order` attribute is used for sorting, while `exclude` is for excluding some information from query results. The `exclude` element is enclosed with an optional value for filtering.

Example 4.1 *A parameter is specified as follow:*

`<param show="1" name="v1:emailTo">tuwien.ac.at</param>`

The parameter in this example is defined for extracting information of email addresses sent to recipients in domain of `tuwien.ac.at`*. The variable* $v1$ *contains the meta-information* `emailTo`*.*

- `<source>` *element*: names of desired sources for users' requests will be put here. This element has only one attribute `name` which is an internal name of data sources. This internal name is assigned automatically by the system.

Example 4.2 *A data source is specified as follow:*

`<source name="at.ac.tuwien.slife.feed.email">Email</source>`

Here, the source is specified using two names, the internal name, which is hidden from users, is put in `name` *attribute; and the external name is optional.*

- `<relation>` *element*: contains a constraint of the VQL query. The required attributes of each constraint are: `id=`*"number"*, `param=`*"variable"*, `source=`*"source-name"*, and optional `or=`*"id"*. The `id` is assigned a number, *variable* in the related `<param>` is used in this `param` attribute. Attribute `source` is identified with $source - name$ or left empty in the case of only one data source specified; and `or` is assigned by `id` of another `<relation>` in order to make an OR expression. The operator for the constraint is specified as a value of the `<relation>` element.

Example 4.3 *A constraint is specified as follow:*

`<relation param="v1" source="Email">str:match</relation>`

This is a constraint to form an expression for the query. The expression means to get the email addresses from `Email` *data source where emails are from* `tuwien.ac.at` *domain. The operator* `str:match` *is a pattern comparison.*

4.3 Operators and Expression in VQL

The SemanticLIFE's back-end is organized by using ontologies and RDF enhanced with data typing and a powerful index mechanism. Hence, the supported operators in VQL inherit these features and reflect them in its operators.

4.3.1 Logic Operators

VQL's logic operators consist of AND, OR and NOT. While NOT is defined in each `<param>` element by using the `exclude` attribute, AND and OR operators are identified in `<relation>` items. OR is defined by the `"or"` attribute, e.g. `or="2"` means that the current constraint will be combined with the constraint number "2" using logic OR operator. AND is implied in a relation without `"or"` attribute.

4.3.2 Comparison Operators

The comparison operators are used in `<relation>` part of VQL queries. These operators are presented as follow:

equal An equal comparison in form of triples which is used by leaving the `<relation>` item empty.

gt The operator means `GREATER-THAN` which is used for comparing values of basic data types such as String, numbers.

lt Similarly to the previous one, the operator means `LESS-THAN`.

dt:gt This operator is used for comparing the date/time value `AFTER` a point of time. The value format of this type conforms XML Schema (XSD)[e] data types, i.e. `'DD-MM-YYTHH:MM:SSZ'` where `'T'` is the delimiter and `'Z'` is the time-zone.

dt:lt Similarly to the `dt:lt` operator; but it means for the date/time value `BEFORE` a point of time.

str:match This is the pattern matching operator for strings. This comparison operator uses common pattern expressions.

ft:match This is the powerful operator relying on the RDF full-text index mechanism applied in SemanticLIFE's metastore. It is used for searching the full-text data such as content of a file or email attachments, or stored WWW pages.

4.3.3 Forming Expressions

Constraints for VQL queries are specified in `<relation>` elements. In order to formulate the expressions for the query criteria, VQL uses `"or"` attribute in `<relation>` elements as boolean OR operator to combine these constraints first, and then it uses boolean AND to combine into the final expression. The sequence of `<relation>` elements are important in combining expressions.

The process of forming expressions is performed with references to the specified sources and ontologies of the systems. A mapping is carried out to map properties in VQL query from the context ontology to the system ontology for RDF queries.

[e]XML Schema, http://www.w3.org/XML/Schema

4.4 The VQL Query Types

4.4.1 VQL Query Types

Definition 2 VQL query types *are different forms of VQL query documents used for special purposes of information retrieval in the VQS.*

Due to the results expected from metastore we distinguish 3 type of queries: *data query type*, *schema query type* and the *embedded query types.* The VQL query types conform to the main syntax of the VQL query document but having some particular parts for each types.

The VQL query types reflect the completeness of itself in supporting users and components fulfil their requests. Data query type is almost used for the information retrieval, while the schema query type is used for tasks related to ontologies such as finding the transitive closure of concepts, retrieving sub hierarchy of concepts. The embedded queries are designed to adapt RDF query experienced user that they can pass their RDF query statements directly for the query process.

4.4.2 Data Query Type

The VQL data query type is commonly used for information querying. A query of this type consists of the four parts as discussed above. In order to inform the VQL parser to process the query as a VQL data query, we identify the `"data"` term in the query: `<query type="data">`.

An example for this query type is shown in Fig. 5: the query retrieves messages' timestamp of files uploaded and browsed Web pages in the SemanticLIFE metastore in `'November 2005'`.

From this query type, we formulate deductive queries for special operations in semantically information retrieval. These operations help users easily get information of interest and obey the principle of VQL design: *"ask minimum words, get maximum information"*. The deductive queries, so-called *VQL Query Operators*, are described in details in Section 5.

4.4.3 Schema Query Type

The syntax of this query type consists of two parts: the first one is the `<statement>` element containing a RDF query statement; and the second part is used to set the query results format. Necessarily, `"schema"` must be identified in the query body. A schema query example is illustrated in Fig. 6.

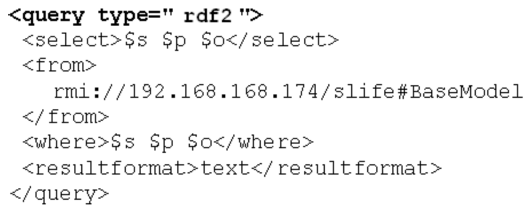
```
<query type="schema">
 <statement>
   select $s
   from &amp;lt;rmi://192.168.168.174/OntologyModel&amp;gt;
   where $s &amp;lt;rdfs:subClassOf&amp;gt;
   &amp;lt;slifeont:FileUploadData&amp;gt; ;
 </statement>
<resultformat>text</resultformat>
</query>
```

Fig. 6. An Example of the VQL SCHEMA Query.

However, the question is, how does the user create RDF query statements? Actually, the schema or ontology queries are offered to clients in the form of query templates or programmatic VQL API.

4.4.4 Embedded Query Types

Similarly, with *embedded query types* RDF query statements are wrapped in VQL query documents. We distinguish two ways of embedding the RDF statements: firstly, a whole statement is embedded; while their parts is embedded separately in the second type.



```
<query type=" rdf2 ">
 <select>$s $p $o</select>
 <from>
   rmi://192.168.168.174/slife#BaseModel
 </from>
 <where>$s $p $o</where>
 <resultformat>text</resultformat>
</query>
```

Fig. 7. An Example of a VQL Embedded Query type 2.

The format of the first embedded query type, so-called VQL RDF query type 1, is similar to the schema query described in Fig. 6, where the term `"rdf1"` is used instead of `"schema"` in the `<query>` tag. Meanwhile, in VQL RDF query type 2, each part of the RDF query statement, such as `select` and `from`, is put in respective query parts.

Fig. 7 is an example of VQL RDF query of type 2, in which `"rdf2"` is specified in `<query>` tag. As depicted in the figure, the expressions of clauses of the RDF query statement are filled in respective elements of the VQL query.

4.5 VQL Query Results Format

In order to increase the flexibility in query results processing, VQL provides five query results formats that are XML format, text format, "jason", RDF graph, and serialized objects of query results. In a VQL query, we identify the query results format in the `<resultformat>` element at the second level of the query document.

4.5.1 Query Results XML Format

The VQL query results XML format is similar to a W3C's format for SPARQL XML query results presented in [14].

This XML format for VQL query results has two main parts: the first one is the list of the query's variables which actually are properties mentioned in the query and additional information such as the message's URI and the data source type of the resource. The second part contains found items, i.e. values of these variables. A VQL query result is described in Fig. 8 is an example.

In order to receive the query result in the XML format, the `"xml"` term should be put in the result-format element of the query document: `<resultformat>xml</resultformat>`.

4.5.2 Query Results Text Format

Some applications tend to avoid using an XML parser in order to simplify processing the query results. Therefore, the text format of query results is an alternative solution for them.

The text format of VQL query results is structured as follows: firstly, a summary of the result is presented, including a number of returned rows and a number of the query's variables; and they are presented as `Rows = num_of_rows` and `Vars = num_of_vars`. In the

```
<answer>
  <query>
    <variables>
      <Datasource/>
      <fileName/>
      <fileLastModified/>
    </variables>
    <solution>
      <Datasource resource="http://www.slife.at/DS#FileUpload"/>
      <fileName>14_07_04_Butler.pdf</fileName>
      <fileLastModified>2004-07-14T15:10:29+02:00
      </fileLastModified>
    <solution>
    ...
  </query>
</answer>
```

Fig. 8. The XML VQL Query Result.

second part, the variable's values of each item are then paired in form of `VAR_NAME = VALUE`. These pairs are connected by semicolons, and one row is for each items. A simulated text format query results of a query is presented in Fig. 9 as follow:

```
Rows = 2
Vars = 3
[ VAR1 = MSG_URI1; VAR2 = VAL21; VAR3 = VAL31 ]
[ VAR1 = MSG_URI2; VAR2 = VAL22; VAR3 = VAL32 ]
```

Fig. 9. The TEXT VQL Query Result.

The VQL query results will be transformed into the text format, if the `"text"` term must be identified in the result-format element of the query document as follows
`<resultformat>text</resultformat>`

4.5.3 Query Results Jason Format

Jason is the common misspelling of JSON[f] and used for a query result format. JSON is a light-weight data-interchange format. It is easy for humans to read and write, for machines to generate and parse. JSON is based on a subset of the Javascript programming language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

In order to request the VQL to return the query result in the JSON format, the `"jason"` term must be put in the result-format element of the query document as
`<resultformat>jason</resultformat>`. Fig. 10 is an example of query results in the JSON format.

[f]JavaScript Object Notation, http://www.json.org/

```
{"head":
   { "vars": ["a" ,"b" ] },
 "results": {
   "distinct": false,
   "ordered": false,
   "bindings": [
   {
     "a": { "type": "uri" , "value": "http://www...." } ,
     "b": { "type": "uri" , "value": "http://www..." }
   } ,
   ...  ] }
}
```

Fig. 10. The JASON VQL Query Result.

4.5.4 Query Results RDF Format

The RDF-graph format of query results is designed for Semantic Web client applications preferring semantic enriched data. Since the RDF format is the standard for this purpose, the query results will be transformed to RDF graphs before returning them to the clients. RDF/XML is used as default RDF format of these query results. Obviously, the clients must have abilities to process RDF-graphed query results by using RDF parsers such as Java-based Semantic Web frameworks, Jena[g] or JRDF[h].

In order to request the VQL returns the query result in form of RDF graphs, the `"rdf"` term must be put in the result-format element of the query document as `<resultformat>rdf</resultformat>`.

4.5.5 Serialized Query Results Object

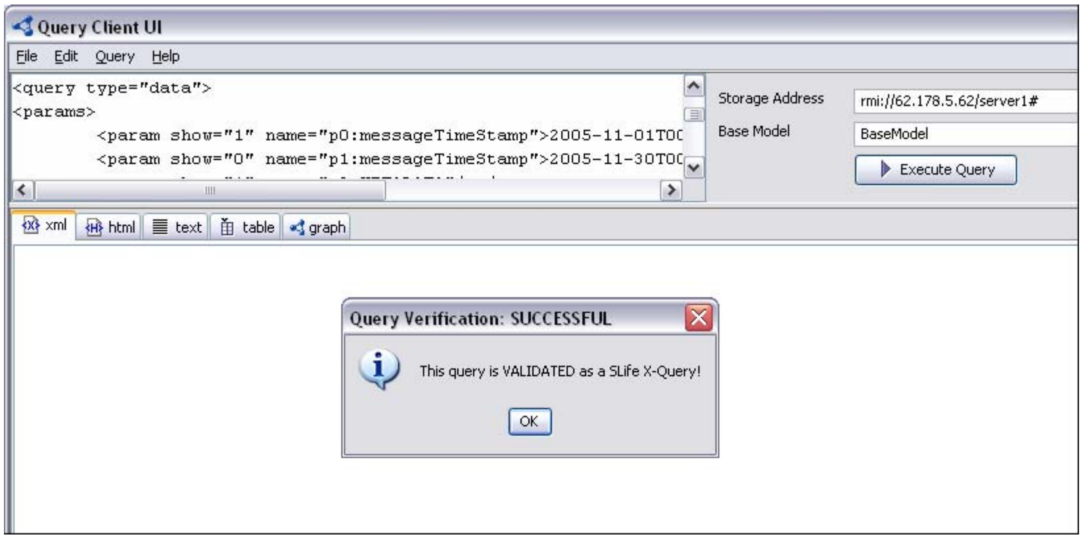
Concerning communication of the internal components of the SemanticLIFE system, sometimes we would like to use the query results object without format transformation. In this case, the VQL must satisfy the demands by support of serializing the results object using the Java serialization technique and sending the serialized object back to the asking component.

In order to request the VQL returning the query result in the form of serialized objects, the `"object"` term must be identified in the result-format element of the query document as `<resultformat>object</resultformat>`. Nevertheless, this format is API-dependent, therefore it is required that the asking component must use the same RDF API as the VQL parser.

4.6 Well-formed and Validated VQL Queries

Checking the validity of the generated VQL queries plays an important in order to help the user dealing with the virtual queries formulation. As first mentioned in Fig. 4 in Section 4.2, we have XML schemas for validating the VQL queries. Actually, there are XML schemas for checking the well-formedness for the VQL query types. Users can choose to verify their VQL queries before executing them; or let the VQS verify them before carrying out any VQL queries.

[g]Jena Semantic Web Framework, http://jena.sourceforge.net/
[h]Java RDF, http://jrdf.sourceforge.net/



```
<query type="data">
<params>
        <param show="1" name="p0:messageTimeStamp">2005-11-01T0C
        <param show="0" name="p1:messageTimeStamp">2005-11-30T0C
```

Storage Address: rmi://62.178.5.62/server1#

Base Model: BaseModel

This query is VALIDATED as a SLife X-Query!

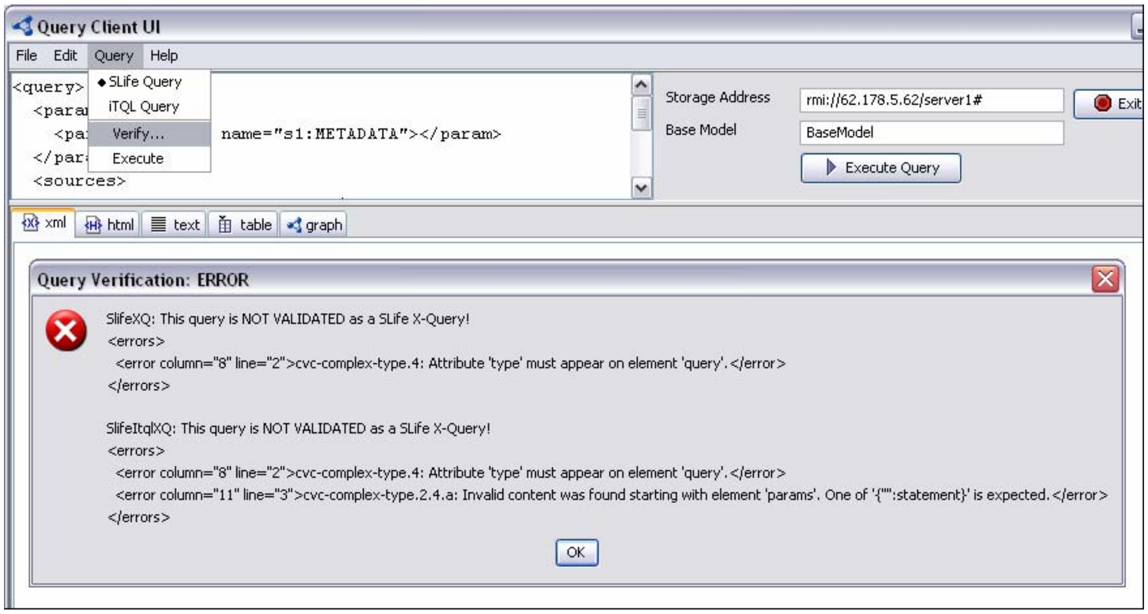
Fig. 11. The VQL query is well-formed and validated.

In general, in VQL we have two formats for VQL queries which are VQL data queries and VQL RDF-embedded queries. Therefore, we have created two XML schemas for each: the first schema is described in Fig. 4 for the data queries; and the second one, whose XML code is presented in Fig. 12, is for the embedded queries.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="query">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="statement"/>
        <xs:element ref="resultformat"/>
      </xs:sequence>
      <xs:attribute name="type" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="statement" type="xs:string"/>
  <xs:element name="resultformat" type="xs:NCName"/>
</xs:schema>
```

Fig. 12. The VQL Query Schema.

VQS uses these schema for validating every VQL queries before executing them. If the VQL query is not validated and well-formed, the system will guide the user to correct it by pointing out where the errors are, as in Fig. 13. In case the user did not specify the type of his VQL query, the system would consequently check out both VQL schemas.



```
<query>
  <para
    <pa                name="s1:METADATA"></param>
  </par
  <sources>
```

```
SlifeXQ: This query is NOT VALIDATED as a SLife X-Query!
<errors>
  <error column="8" line="2">cvc-complex-type.4: Attribute 'type' must appear on element 'query'.</error>
</errors>

SlifeItqlXQ: This query is NOT VALIDATED as a SLife X-Query!
<errors>
  <error column="8" line="2">cvc-complex-type.4: Attribute 'type' must appear on element 'query'.</error>
  <error column="11" line="3">cvc-complex-type.2.4.a: Invalid content was found starting with element 'params'. One of '{"":statement}' is expected.</error>
</errors>
```

Fig. 13. The VQL query is not well-formed.

If the query is well-formed, the system could carry it out. Fig. 11 shows an example of validating a well-formed VQL query, the information will be shown to the user.

5 VQL Query Operators

In this section, we present *VQL Operators* which are deductive queries from the data query type for special operations. These are used for making complex queries in a simpler manner which helps users achieve *"minimum of words, maximum of information"*. This conforms to the principle of building user-centered applications as in [15].

5.1 GetInstances Operator

GetInstances operator is the common form of VQL data queries. The operator retrieves appropriate information according to the criteria described in parameters, sources, constraints of the query. The properties with `show` attribute set to `"1"` will be included into the query results.

Fig. 5 in Section 4 is an example of *GetInstances* operator. As depicted, the query is about retrieving the message's time-stamp from 01/11/2005 to 30/11/2005 of uploaded files and browsed WWW pages. The operators in the constraint part, `"dt:gt"` and `"dt:lt"`, are combined using boolean `'AND'`.

5.2 GetInstanceMetadata Operator

This query operator assists the user easily retrieve all metadata properties and their values of resulting instances. This query operator is very useful when the user does not care or not know exactly what properties of the data instances are available. This could happen when he/she makes a request; or he/she would like to get all metadata of these data items in the simplest way.

In order to make a *GetInstanceMetadata* operator, we must put one parameter in the query document with the reserved string `"METADATA"`. The other parameters could be used for the criteria of the query to filter the query results. The rest of the query document is similar to a normal data query. Fig. 14 describes a example of this operator.

```
<query type="data">
 <params>
  <param show="1" name="p0:messageTimeStamp">2005-11-01T00:00:00</param>
  <param show="1" name="p1:METADATA"/>
 </params>
 <sources>
  <source name="fileupload">FileUpload</source>
 </sources>
 <relations>
  <relation id="1" param="p0" source="">dt:gt</relation>
 </relations>
 <resultformat>xml</resultformat>
</query>
```

Fig. 14. VQL GetInstanceMetadata Query Operator.

Here, the query is about getting the *metadata* and their values of uploaded files which are sent from 01/11/2005, as well as the timestamps of these files.

5.3 GetRelatedData Operator

In Semantic Web applications, particularly in the SemanticLIFE system, finding relevant or associated information plays an important role. When we make a query to search for a specific piece of information, we also would like to see associated information to what we found. In other words, we expect to retrieve the related

```
<query type="data">
 <params>
  <param show="0" name="p1:emailTo">hta@gmx.at</param>
  <param show="1" name="p2:RELATED-WITH"/>
 </params>
 <sources>
  <source name="email">Email</source>
  <source name="contact">Contact</source>
 </sources>
 <relations>
  <relation id="1" param="p1" source="email"/>
 </relations>
 <resultformat>xml</resultformat>
</query>
```

Fig. 15. VQL GetRelatedData Query Operator.

For example, when we are looking for an email message with a given email address, we also want to see the linked data to this email such as the contact having this email address, appointments of the person in the email and/or Web pages browsed by this person. Obviously, this operator shows the VQL power and obeys the principle of *"minimum words, maximum information"* for users.

In order to make a request of this operator, we must identify a parameter containing the a reserved word `"RELATED-WITH"` in the query document. The `<sources>` element is used to limit a range of data sources in searching associated information as presented in Fig. 15.

In this example, the query asks for instances of `Email` data source containing a specific email address, e.g. `hta@gmx.at`; and from found messages. Rhe related information in the appropriate data sources, which are identified in `<sources>` of the query, will be located as well. The found associated data will be used as a recommended results for the user.

5.4 GetLinks Operator

This query operator operates by using the system's ontology and RDF graph pattern traversal. The operator aims at finding out the associations/links between instances and objects. For instance, we are querying for a set of instances of emails, contacts and appointments, and normally, we receive these data separately. However, what we are expecting here is that the links between instances (as well as objects) are also provided. The links are probably properties of email addresses, name of the persons, locations, and so on.

The *GetLinks* operator helps us to fulfill this expectation. This operator is similar to the *GetInstanceMetadata* operator in the way of exploiting the metastore. While *GetInstance−Metadata* operator tries to get related instances based on analysis of a given link or information and the ontologies, *GetLinks* extracts the associations in instances or objects. In order to make a *GetLinks* operator, the reserved word `"SLINKS"` (*semantic links*) must be identified in one `<param>` element.

```
<query type="data">
 <params>
  <param show="1" name="p1:emailTo">hta@gmx.at</param>
  <param show="1" name="p2:conName">Hoang Thieu Anh</param>
  <param show="1" name="p3:SLINKS"/>
 </params>
 <sources>
  <source name="email">Email</source>
  <source name="contact">Contact</source>
  <source name="calendar">Calendar</source>
 </sources>
 <relations>
  <relation id="1" param="p1" source="email" or="2"/>
  <relation id="2" param="p2" source="contact"/>
 </relations>
 <resultformat>xml</resultformat>
</query>
```

Fig. 16. VQL GetLinks Query Operator.

We distinguish the links between instances and metastore objects as following: if sources are specified in the query document without any parameters except `"SLINKS"`, then the links will be detected between objects. Otherwise, if some parameters are shown, the links are implied for instances' associations. An example of *GetLinks* query operator is described in Fig. 16. The query will return the associations between instances having the given receiver's email and the contact name in three data sources `Email`, `Contact`, and `Calendar`.

By providing these operators, VQL offers a powerful feature of navigating the system by browsing data source by data source, instances by instances based on found semantic associations.

5.5 GetFileContent Operator

The SemanticLIFE system covers a large range of data sources, from personal data such as contacts, appointments and emails to files stored on a computer, e.g. office documents, PDF files, media files. Therefore, a query operator to get the contents of these files is necessary.

```
<query type="data">
 <params>
  <param show="0" name="p0:filePath">
   c:/slifedata/uploadedfiles/2006/01/15/CFP_WISM_06.pdf
  </param>
  <param show="1" name="p1:CONTENT"/>
 </params>
 <sources>
  <source name="fileupload">FileUpload</source>
 </sources>
 <relations/>
 <resultformat>xml</resultformat>
</query>
```

Fig. 17. VQL GetFileContent Query Operator.

Normally, to carry out this task, we must define two parameters in the query document, the first one is the file path retrieved from the previous query; and the second one is defined with reserved word `"CONTENT"`. In the `<source>` element of the query, a data source is identified as a reference; and the constraint part is often left empty. The query, described in Fig. 17, is an example of the *GetFileContent* operator, where the content of the file having name `"CFP_WISM_06.pdf"` with its full path will be extracted from the metastore.

The files have been uploaded to SemanticLIFE metastore through a Message Handler [2] (Service Bus), and they are coded using BASE64 data encoding[i]. The Analysis component will decode and store them in the file system, then it carries out a full-text index on these files for faster and more accurate retrieving later on. After executing the *GetFileContent* query, clients receive a BASE64-encoded string encapsulated in a XML document. For the user, what he/she needs is the 'real' contents of the file, and it is the VQS interface's turn to do the job that decodes the content and shows it to the user [1].

6 VQL Parser: VQL - RDF Query Languages Mapping

The VQL parser translates the VQL query to correspondent RDF query statements; accesses the metastore to get results, then transforms them to the appropriate format and sends the processed results back to the user. In this section, we present the first stage of a query process in the VQS [1] using VQL that is the mechanism to map VQL queries to RDF queries. The mappings consist of three phases: expressions mapping, syntax mapping; and semantic mapping.

[i] RFC-3548, http://www.faqs.org/rfcs/rfc3548.html.

6.1 Expression Mapping

Expressions in VQL queries are likely to be mathematical ones, while the "expressions" in RDF queries are in form of the triples. Therefore, an accurate expression mapping from normal expressions in VQL queries to triple expressions in RDF queries is crucial. VQL carries out this task as follows:

1. Forming mathematical expressions from elements of a VQL query;
2. Representing the final aggregated expression into an expression tree;
3. From the expression tree, we traverse and formulate triple expressions for RDF query by references to ontologies and related sources in the VQL query.

Example 6.1 *Taking the listing in Fig. 5 as an example, an expression will be formed from the described query's parts as follows (here $msgTS$ is used instead of $messageTimeStamp$ for shorter representation):*

$$(msgTS \geq \#01/11/2005\#) \cap (msgTS \leq \#30/11/2005\#)$$

From this expression, we can create the expression tree as depicted in Fig. 18.

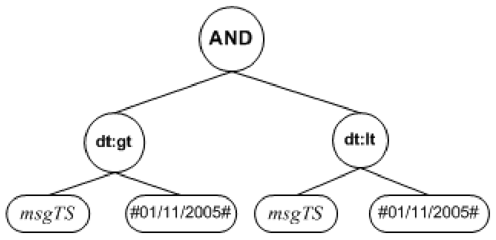


Fig. 18. The expression tree for Example 6.1.

Using this tree, we generate the triple expressions for the RDF query by traversing the tree. The generated triple expressions are described below in an RDF query language:

```
(<slife:msgTS> <slife:after> '2005-11-01T00:00:00Z') and
(<slife:msgTS> <slife:before> '2005-11-30T00:00:00Z')
```

where, `"slife"` is the namespace for the ontology and data schema of SemanticLIFE metastore; and `<slife:after>` and `<slife:before>` are comparison operators of the VQL.

Furthermore, the data sources are taken into account during mapping expressions. The specified data sources in the VQL query document (in `sources` part and `<relation>` elements) are used for either identifying the expression applied on them or generating correspondent queries for them. Hence from a VQL query, more than one RDF query is probably generated.

6.2 Syntax Mapping

Syntax mapping takes care of the issue of translating from VQL query syntax to a RDF query language syntax. VQL queries are actually interpreted as `SELECT` statements of RDF query languages. The iTQL's `SELECT` statement contains three required clauses `select`, `from`, `where`, and optional clauses such as `orderby`, and `exclude`. The syntax mapping will parse VQL query's parts to the clauses of the RDF `SELECT` statement(s).

First of all, the `select` clause of the RDF query will be filled by `<params>` parts of the VQL query. The parameters set to `"1"` will be put as variables of the `select` clause, and hidden parameters (set to `"0"`) are used for forming the criteria only. Additionally, some extra variables will be added in the clause to get the message's URI and the name of data sources. Secondly, the `from` clause will be generated automatically by the VQL parser. For the time being, all data sources are stored in one huge metastore with a unique network address. And this network address plugged access protocol will be placed in the `from` clause. Thirdly, generated triple expressions will be used for the `where` clause. As discussed, the process of generating the triple expression combines all three parts of the VQL query - `params`, `sources`, and `relations` - along a further analysis by adding more expressions to clarify the criteria.

Last but not least, we have two optional attributes, `order` and `exclude`, for each parameter. If the `order` is set to `"1"`, then the variable will be added into `orderby` clause. And if an `exclude` is specified, an expression in form of `exclude(?param ?op ?exclstr)` will be added into the `where` clause.

6.3 Semantic Mapping

The semantic mapping takes part in both mapping tasks above to resolve semantic ambiguity problems. This is the vital and decisive mapping task of the VQL. The semantic mapping is going to solve the following concerns during the query generating process from the user's initial VQL query:

- Disambiguating query items;
- Aligning querying and queried concepts;
- Resolving semantic conflicts.

6.3.1 Disambiguating Query Items

The inaccuracy of a query is mainly due to the ambiguities inside itself. Coping with ambiguous items in the VQL query made by a user is a decisive step in parsing it later on. Here the ambiguity could be in a terminological manner, i.e. requested data properties are not clear. For example, a `"Name"` property in a query is ambiguous because the query parser can not identify which “name” will be extracted, contact name or name of email sender/receiver.

Clarifying these properties could be done by using the data sources specified in the query and the ontologies of the system. Based on the data sources, the appropriate properties in the ontology will be detected and used instead of ambiguous items. For instance, concerning the `"Name"` property is described above, this mapping task must rely on the related source described either in source or constraints elements, e.g. `Contact`; after on, based on ontologies, appropriate properties will be located such as `contactFirstName`, `contactLastName`.

6.3.2 Concepts Aligning

The concepts aligning is used to map concepts used in VQL queries to appropriate concepts in the system ontology. This mapping is an important step in generating RDF queries containing concepts derived from the system ontology. Moreover, it guarantees the relevant information will be extracted.

For example, my context ontology is about conferences (organizing, participating) containing concepts of contact persons, venues, publication databases, discussion logs, talks, and so on. The system ontology as mentioned is about the personal information such as email messages, chat logs, calendars (appointments), browsed Web pages, and so on. Now, I need the following information from the system: contacted persons who will attend incoming CAiSE '08 conference and the location where the conference is organized (an URL pointing to an interactive digital map is preferable).

The formulated VQL query would contain concepts of `Contact`, `Conference`, `Place`. Next, in order to retrieve necessary information, these concepts are mapped into the system ontology with more detail, i.e. `Contact`, `Conferences`, and `Place` are mapped to related concepts containing the personal information such as personal contacts, email messages, appointments, annotated chat/call logs, annotated browsing Web pages, events and the location information (map, country, city, street and building). Next, appropriate properties will be selected for constraints of time, location and name of the conference.

6.3.3 Resolving Semantic Conflicts

Further disambiguation of users' queries is needed, especially when the user asks for information using an ambiguous entity over many data sources. The issue is how to identify user's "intended" properties for each data source. For example, `"Name"` property is constrained with `Email` and `Contact`. If the `"Name"` property is constrained with a source identified in a `<relation>` element, then the issue is similar to the discussion above. Otherwise, the query has to:

- get all related properties in system ontology based on specified data sources. In this case, there are probably more than one query generated; or
- suggest the most related properties from the ontology based on a *semantic similarity* of properties in the same query. For instance, if other properties are requested for `Contact` items, so that the "names" of `Contact` object would be suggested.

Obviously, the strategies could be combined to deliver an aggregated solution which resolves this problem.

Furthermore, all discussions above are mainly focused on VQL data query; however these solutions of the semantic mapping is applied for all types of the VQL query. Generally, all user-entered queries should be checked for implied semantic problems as well as the syntax of them. All these problems could be limited by using a query formulation graphical user interface of the VQS which is discussed in detail in another context [10]; and the system library (API) as well. Nevertheless, because of the openness of the VQS and VQL, these issues must be taken care.

7 Context-Aware Query Results Representation

The query results come back to the VQS according to the schema in the SemanticLIFE metastore. Therefore, for the user, the results are put into the "context" of the query concerning the VDC's ontology—the *context ontology*—before presenting to the user as depicted in Fig. 19.

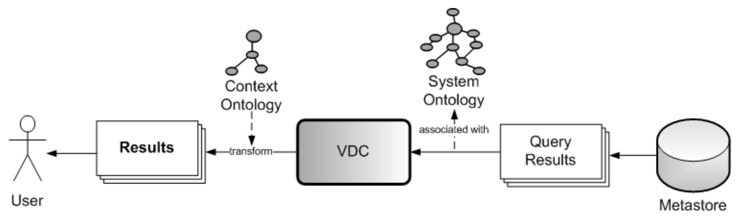


Fig. 19. The Process of Returning Query Results in the VQS.

As described in the figure, the results from the SemanticLIFE metastore are based on the system ontology; therefore, via the VDC, the query results are *contextualized* with the VDC's context ontology. Moreover, based-on this context ontology, the related information to the relationships within its hierarchy is also presented as the recommendation to the user.

For instance, according to the example in Section 6.3.2, the results are associated to the concepts of system ontology, the PIM ontology. A transformation will be carried out at the VDC to render those results to the user's context—context ontology—with concept of `Contact`, `Conference`, `Place` in order to fulfill the user's request: contacted persons and the venue information of CAiSE'08 conference.

8 Conclusion and Outlook

In this paper we have presented the Virtual Query Language, a design of a query language aiming at a significant complexity reduction in formulating semantically meaningful queries.

Along with the VQS, this query language design helps VQS's users comfortably work with the virtual information. With support of the VQL, the users only care about the concepts (information) of interest when they formulate the requests. The VQL also introduces the special operators deducted from the VQL query data type that support users in making complex queries in a simpler manner, as well as the operators that are also used by the VQS itself for further analysis in the information retrieval process.

In addition, in this paper we also presents the parsing mechanism for VQL. The VQL parser can deal with the syntactic mapping from VQL queries to RDF queries. It can cope with the semantic mapping: disambiguating the concepts and semantic conflicts.

As the next steps of VQL development, we will consider RDF as an alternative format for VQL, and building a graphical interface on top of this language that keeps the transparency of the language to users.

Acknowledgments

This work has been generously supported by ASEA-UNINET. It is a dedication to the SemanticLIFE-Team for their nice contribution in the success of the SemanticLIFE framework development.

References

1. H. H. Hoang, A. Andjomshoaa, and A M. Tjoa (2006), *VQS: An ontology-based query system for the SemanticLIFE Digital Memory project*, In 2th IFIF WG 2.14 & 4.12 International Workshop on Web Semantics (SWWS 2006), Vol. 4278, Lecture Notes of Computer Science, Springer, pp. 1796–1805.
2. M. Ahmed, H. H. Hoang, S. Karim, S. Khusro, M. Lanzenberger, K. Latif, E. Michlmayr, K. Mustofa, T. H. Nguyen, A. Rauber, A. Schatten, T. M. Nguyen, and A M. Tjoa (2004), *SemanticLIFE - A framework for managing information of a human lifetime*, In 6th International Conference on Information Integration and Web-based Applications and Services (iiWAS 2004), OCG Book Series, pp. 725–734.
3. N. Athanasis, V. Christophides, and D. Kotzinos (2004), *Generating on the fly queries for the Semantic Web: The ICS-FORTH Graphical RQL interface (GRQL)*, In 3rd International Semantic Web Conference (ISWC 2004), Vol. 3298, Lecture Notes of Computer Science, Springer, pp. 486–501.
4. T. Catarci, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris (2004), *An ontology based visual tool for query formulation support*, In On The Move Federated Conferences (OTM 2004) Workshops, Vol. 2889, Lecture Notes of Computer Science, Springer, pp. 32–33.
5. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl (2002), *RQL - A declarative query language for RDF*, In 11th International World Wide Web Conference (WWW 2002), ACM Press, pp. 592–603.
6. P. Gray, K. Hui, and A. Preece (2001), *An expressive constraint language for Semantic Web applications*, In 7th International Joint Conference on Artificial Intelligence (IJCAI 2001) - Workshop on E-Business and the Intelligent Web, pp. 46–53.
7. R. Guha and R. McCool (2003), *TAP: a Semantic Web platform*, *Int. J. Computer and Telecommunications Networking*, Vol. 42, No. 5, ACM Press, pp. 557–577.
8. A. Andjomshoaa and K. Latif (2006), *Service-Oriented Pipeline Architecture*, JAX Innovation Award 2006 Proposal, `http://jax-award.de/jax_award06/proposal_view_en.php?id=55`.
9. H. H. Hoang, A. Andjomshoaa, and A M. Tjoa (2006), *Towards a new approach for information retrieval in the SemanticLIFE Digital memory framework*, In 6th IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006), IEEE Computer Society, pp. 485–488.
10. H. H. Hoang, T. M. Nguyen, A. Andjomshoaa, and A M. Tjoa (2006), *A front-end approach for user query generation and information retrieval in the SemanticLIFE framework*, In 8th International Conference on Information Integration and Web-based Applications and Services (iiWAS 2006), OCG Book Series, pp. 107-116.
11. A. Seaborne (2004), *RDQL - a query language for RDF*, W3C Member submission 9 January 2004.
12. E. Prud'hommeaux and A. Seaborne (2007), *SPARQL query language for RDF*, W3C Recommendation 12 November 2007.
13. D. Wood, P. Gearon, and T. Adams (2005), *Kowari: A platform for semantic web storage and analysis*, In XTech 2005 Conference, `http://www.idealliance.org/proceedings/xtech05/papers/04-02-04/`.
14. D. Beckett (2007), *SPARQL query results XML format*, W3C Recommendation 12 November 2007.
15. E. Garcia and M.-A. Sicilia (2003), *User interface tactics in ontology-based information seeking*, *Psychology Journal*, Vol. 1, No. 3, pp. 242–255.