The Virtual Query Language for Information Retrieval in the SemanticLIFE Framework*

Hanh Huu Hoang and A Min Tjoa

Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188
A-1040 Vienna, Austria
+43 1 58801 18861
{hanh, tjoa}@ifs.tuwien.ac.at

Abstract. Creating an innovative mediator environment such as graphical user interfaces, lighter weight languages to help the users in the challenging task of making unambiguous requests is crucial in semantic web applications. The query language in the Virtual Query System (VQS) of the SemanticLIFE framework is designed for this purpose. The proposed query language namely Virtual Query Language (VQL) is in a further effort of reducing complexity in query making from the user-side, as well as, simplifying the communication for components of the SemanticLIFE system with the VQS, and increasing the portability of the system.

1 Motivation

Making unambiguous queries in the Semantic Web applications is a challenging task for users. The Virtual Query System (VQS) [7] of the SemanticLIFE framework [1] is an attempt to overcome this challenge. The VQS is a *front-end* approach for user-oriented information retrieval.

The issue is in the user-side, where users are required to make queries for their information of interest. The RDF query languages are powerful but they seem to be used for back-end querying mechanisms. To users, who are inexperienced with them, these query languages are too complicated to use. Additionally, the communication of components of the SemanticLIFE system with the VQS requires the facility to transfer their requests without knowledge of the RDF query language. Another important point is the portability of the system, i.e. if the system is bound to specific RDF query language, we could have problems when shifting to another one.

In the effort of coming over the above tackles, we, in this paper, present a new query language namely *Virtual Query Language* (VQL). The language is used by the VQS for information querying in the SemanticLIFE system. The

* This work has been generously supported by ASEA-UNINET and the Austrian National Bank within the framework of the project Application of Semantic-Web-Concepts for Business Intelligence Information Systems - Project No. 11284.

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 is briefly presented in Section 2. Next, details of the VQL are pointed out in Section 3. Section 4 introduces VQL query operators; and the issues of mapping a VQL query to the respective RDF query are then described in Section 5. Finally, the paper is concluded with a sketch of the future work.

2 Related Work

There are two main approaches to reduce the difficulty in creating queries from user-side in Semantic Web applications. The first trend is going to design the friendly and interactive query user interfaces to guide users in making the queries. The high-profiled examples for this trend are GRQL [2] 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 [8]. 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 pre-prepared domain-specific patterns to choose from as a starting point, which he can then extend and customize. 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 neighbourhood of the currently selected class.

The second approach of reducing complexity is the effort in 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 [10] are hight-rate examples.

[6] describes an 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 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 [10] of TAP[1] expresses 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

[1] TAP Infrastructure, http://tap.stanford.edu/.

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.

Our approach, VQL, continues this trend to design an effective and lighter weight query language to assist the users make queries in simple manner and simplify the communication between components of the SemanticLIFE system with the query module - the VQS.

3 The Virtual Query Language - VQL

3.1 The Goals of the VQL

A number query languages have been developed for the Semantic Web data such as data in form of RDF (RQL [8], RDQL [11], SPARQL [9], and iTQL [12]). Why do we need yet another query language?

These query languages all 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 to construct queries to normal users 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 which supports "semantic" manner from users' queries. In the context of the VQS and SemanticLIFE system, we can see the aims of VQL as follows:

- VQL helps clients making queries without knowledge of RDF query languages. The user just gives basic parameters of needed information in VQL queries, and would receive the expecting results.

- VQL assists users in navigating the system via semantic links or associations provided in the powerful query operators based on ontologies.

- VQL simplifies the communication between Query module and other parts. Since the components asking for information do not need to issues the RDF query statement, which is uneasy task for them. As well as this feature 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 the 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 systems, especially the interface of the system database.

3.2 The Syntax of VQL

Query Document Syntax. Generally, a VQL query is a document having four parts: parameters, data sources, constraints, and query results format as the schema depicted in Fig. 1.

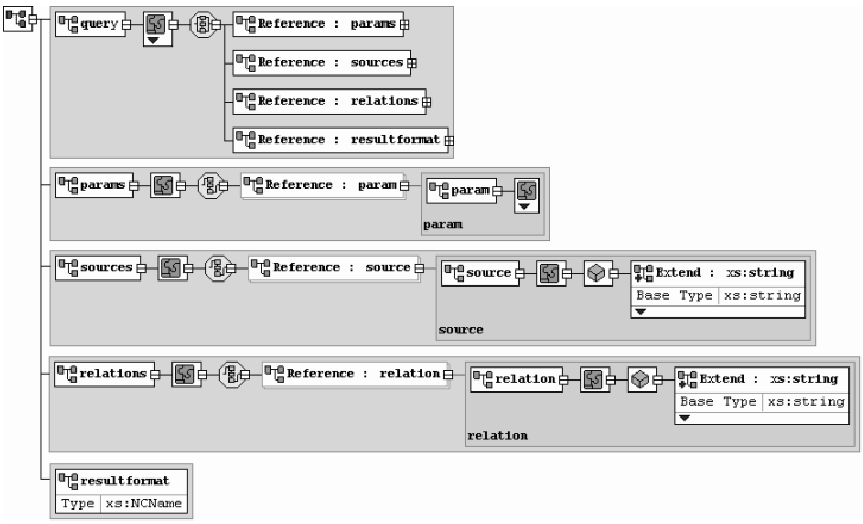

Fig. 1. The schema for general VQL queries

The first part contains parameters of 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 is used for specifying the sources where the information will be referred to extract from. Obviously, the information need defined in the first part must be related to the sources specified in this part.

Thirdly, the constraints of the query 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 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.

XML-based Format. A standard format for information exchange, a 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 follow (Fig. 2):

- The top level or the body of query is `<query>` element. Here, the type of the query must be specified in `type` attribute. The reserved terms used for this attribute are `"data"`, `"schema"`, `"itql1"` and `"itql2"` for different VQL query types (see section 3.4). 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

```
<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. 2. An XML-based VQL Query Example

`<params>`, `<sources>`, and `<relations>` are used once for each. These elements have their children specified in the third level. Fig. 2 is an example. For the schema query or the iTQL type 1 query, we use only one `<statement>` element which contains a RDF query statement (Fig. 3). For the iTQL2 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. 4. Moreover at this level, we must identify the query results format in the `<resultformat>` element by a term among `"xml"`, `"text"`, `"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 element is enclosed with an optional value for filtering.

- `<source>` *element*: names of concerned 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.

- `<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 put as value of the `<relation>` element.

3.3 Operators and Expression in VQL

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.

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. Value format of this type confronts XML Schema (XSD) data types.

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 expression.

ft:match This is the powerful operator relying the full-text index 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.

Expressions. 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.

3.4 The VQL Query Types

Data Query Type. 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 VQL parser to process the query as a VQL data query, we identify `"data"` term in the query: `<query type="data">`.

An example for this query type is shown in Fig. 2: the query retrieves messages' time-stamp of files uploaded and browsed web pages in the SemanticLIFE repository in November 2005.

From this query type, we obtain 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 less, get more."* The deductive queries, so-called VQL query operators, are detailed in Section 4.

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. 3.

```
<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. 3. An example of VQL SCHEMA query

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

iTQL Query Types. Similarly, with 'iTQL query types' RDF query statements are wrapped in VQL query documents. The name of 'iTQL query type' comes from the RDF query language used in the system. Since the SemanticLIFE framework uses Kowari [12] as its back-end, so that iTQL RDF query language is used for query statements. We also 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="itql2">
 <select>$s $p $o</select>
 <from>
    rmi://192.168.168.174/slife#BaseModel
 </from>
 <where>$s $p $o</where>
 <resultformat>text</resultformat>
</query>
```

Fig. 4. An example of a VQL iTQL query type 2.

The format of the first iTQL query type, so-called VQL iTQL query type 1, is similar to the schema query depicted Fig. 3, where the term `"itql1"` is used instead of `"schema"` in the `<query>` tag. Meanwhile, in VQL iTQL query type 2, each parts of RDF query statement, such as `select` and `from`, are put in respective query parts. With the iTQL's `SELECT` statement, its parts are: `select`, `from`, `where`, `orderby`, `limit`, and `offset`.

Fig. 4 is an example of VQL iTQL query of type 2, in which `"itql2"` 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.

3.5 Query Results Format.

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

Query Results XML Format: `<resultformat>xml</resultformat>`

VQL query results XML format is similar to a W3C's format for SPARQL query results presented in [3]. This XML format has two main parts: the first one is the list of the query's parameters and needed additional information. The second part is the list of found instances.

Query Results Text Format: `<resultformat>text</resultformat>`

The text format of VQL query results is structured as following: the variable and its value 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.

Query Results RDF Format: `<resultformat>rdf</resultformat>`

The RDF-graph format of query results is designed for semantic web client applications, here they prefer semantic enriched data. The query results will be transformed to RDF graphs before returning them to the clients.

Serialized Query Results Object: `<resultformat>object</resultformat>`

Concerning with the inside components communication in the SemanticLIFE system, components sometimes would like to use query results object without format transformation. The VQL serializes the results and send the serialized objects back to the asking component.

4 Query Operators of VQL

In this section, we present deductive queries for special operations in making complex queries in simpler manner which helps users *"ask less, get more."* This is the principle of building user-centered applications [5].

4.1 GetInstances Operator

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

Fig. 2 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'`.

4.2 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 data instances are. The case could happen when he/she makes request; or he/she would like to get all metadata of these data items by 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. 5 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. 5. 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.

4.3 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. 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, web pages browsed by this person. Obviously, this operator shows the VQL power and obeys the principle of *"ask less, get more"* 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 following figure (Fig. 6).

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, the related information in the appreciate data sources, which are identified in `<sources>` of the query, will be located as well.

```
<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. 6. VQL GetRelatedData Query Operator.

4.4 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) provided additionally. The links are probably properties of email addresses, name of the persons, locations, and so on.

GetLinks operator helps us to fulfill this expectation. This operator is similar to $GetInstanceMetadata$ in the way of exploiting the metastore. While $GetInstanceMetadata$ 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. 7. VQL GetLinks Query Operator.

We distinguish the detecting links between instances and 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. 7. 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.

4.5 GetFileContent Operator

The SemanticLIFE system covers a large range of data sources, from personal data such as contacts, appointments, emails to files stored in his/her computer, e.g. the office documents, PDF files, media files. Therefore, a query operator to get the contents of these files is very 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. 8. 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 got 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 is described in Fig. 8 is an example of *GetFileContent* operator, where a content of the file having name "CFP_WISM_06.pdf" with its full path will be extracted from the metastore.

5 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 appreciate 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 [7] 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.

5.1 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 following steps:

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 with referring to ontologies and related sources in the VQL query.

Example: Looking back at Fig. 2 as an example, an expression will be formed from 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. 9.

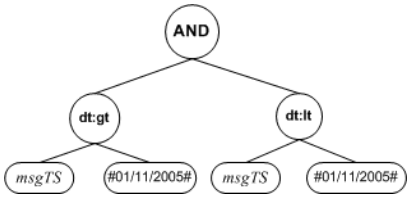

Fig. 9. The expression tree for the Example 4.

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

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

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

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 are probably generated.

5.2 Syntax Mapping

Syntax mapping takes care 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 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 unshown 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 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 `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.

5.3 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 query generating process from the user's initial VQL query:

- Disambiguating query items
- Resolving semantic conflicts

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 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 data source specified in the query and the ontologies of the system. Based on the data sources, the appreciate 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, appreciate properties will be located such as `contactFirstName`, `contactLastName`.

Resolving semantic conflicts. In a further disambiguation users' queries, especially when the user asks for information using an ambiguous entity over many data sources. The issue is that how to identify user's "intended" properties for which data sources. For example, `"Name"` property is constrained with `Email` and `Contact`. If the `"Name"` properties 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 ontology based on a *semantic similarity* of properties in the same query. For instance, if other properties are major requesting for `Contact` items, so that the "names" of `Contact` object would be suggested.

The semantic mapping is applied for all types of the VQL query. Generally, all user-entered queries should be check for implied semantic problems as well as the syntax of them.

6 Conclusion and Future Work

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

As the next steps of VQL, the issues of improving by considering RDF as the alternative format for VQL, and building a graphical interface on top of this language will be discussed consequently in the details. We see the advantages of the RDF as standard for the Semantic Web applications, so that coding the VQL in RDF as 'RDF forms' would be considered. As well as, a powerful interface for VQL in context of the VQS will be taken into account.

References

1. 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. Semanticlife - a framework for managing information of a human lifetime. In *Proceedings of the 6th International Conference on Information Integration and Web-based Applications and Services*, September 2004.
2. N. Athanasis, V. Christophides, and D. Kotzinos. Generating on the fly queries for the semantic web: The ics-forth graphical rql interface (grql). In *Proceedings of the 3rd International Semantic Web Conference*, pages 486–501, 2004.

3. D. Beckett. Sparql query results xml format. Technical report, W3C Working Draft, August 2005.
4. T. Catarci, P. Dongilli, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 308–312, 2004.
5. E. Garcia and M.-A. Sicilia. User interface tactics in ontology-based information seeking. *PsychNology Journal*, 1(3):242–255, 2003.
6. P. Gray, K. Hui, and A. Preece. An expressive constraint language for semantic web applications. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 2001.
7. H. H. Hoang, A. M. Tjoa, and T. M. Nguyen. Ontology-based virtual query system for the semanticlife digital memory project. In *Proceedings of the 4th International Conference on Computer Sciences*, February 2006.
8. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. Rql - a declarative query language for rdf. In *Proceedings of the Eleventh International World Wide Web Conference*, pages 592–603. ACM Press, August 2002.
9. E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. W3C Working Draft, November 2005.
10. R. M. R. Guha. Tap: a semantic web platform. *International Journal on Computer and Telecommunications Networking*, 42(5):557–577, August 2003.
11. A. Seaborne. Rdql - a query language for rdf. Member submission, W3C, 2004.
12. D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *Proceedings of the 14th International WWW Conference*, 2005.