RDF-GL: A SPARQL-Based Graphical Query Language for RDF

Frederik Hogenboom, Viorel Milea, Flavius Frasincar, and Uzay Kaymak

1 Introduction

In an era of ever-increasing information needs, the ability to query large databases quickly and efficiently has come to play a major part. For a large share, this growing need is addressed by tools and languages aimed at performing complex queries on distributed data. However, the intuitiveness of designing such complex queries has only been addressed to a limited extent, making such tools available solely for technical users.

The realm of such tools, aimed at the intuitiveness of query design, though rather limited, presents some interesting applications. Examples of interfaces aimed at the non-technical user include EROS [19] and SPARQLViz [4]. Additionally, several graphical query languages (GQL) enable users to create queries only by arranging and connecting symbols on a virtual canvas. Therefore, complete knowledge of a normal query language is not necessary, as GQL's are mainly focused on intuitiveness of use.

Next to intuitive queries, the representation of knowledge is gaining importance, especially in the context of Web-based applications. New standards are being developed for this purpose under a common denominator - the Semantic Web [3]. One of the state-of-the-art languages put forward by this initiative is the Resource Description Framework (RDF) [5]. The language enables representations centered around the meaning of data, rather than the presentation hereof, and allows the inference of implicit knowledge from explicitly modeled data. The state-of-the-art query language for RDF is SPARQL [14].

Graphical query languages have already been developed for different types of relational representations, but no SPARQL-based GQL is available yet for querying RDF models. Our main focus is to propose one such GQL for RDF,

Econometric Institute, Erasmus University Rotterdam
P.O. Box 1738, 3000 DR Rotterdam, the Netherlands
{fhogenboom, milea, frasincar, kaymak}@ese.eur.nl

based on SPARQL. For this purpose we introduce RDF-GL, our graphical query language for RDF. Additionally, we present SPARQLinG, an application aimed at the design of graphical RDF-GL queries.

After discussing approaches related to our current goal in Sect. 2, we move on to presenting our main contribution, the SPARQL-based graphical query language for RDF, RDF-GL, in Sect. 3. The application developed for designing RDF-GL queries, SPARQLinG, is presented in Sect. 4. We conclude in Sect. 5.

2 Related Work

This section is aimed at providing an overview of current research efforts related to graphical query languages. Although none of the presented approaches is built around RDF and SPARQL simultaneously, we deem some of the ideas presented relevant for the current goals, as outlined in the following sections. A summarizing overview of the main features of the presented graphical query languages is provided in Table 1. The four attributes considered in this overview consist of whether the considered approach i) is a true graphical query language, ii) shows a graphical user interface, iii) the query language on which the tool is based, and iv) the data language for which it is intended.

Table 1 GQL features summary.

	GQL	GUI	Query language	Data language
DERI	yes	no	–	RDF
XML-GL	yes	no	–	XML
GLOO	yes	no	nRQL	OWL ontologies
EROS	no	yes	RQL	RDFS
SPARQLViz	no	yes	SPARQL	RDFS
GRQL	no	yes	RQL	RDFS
SEWASIE	yes*	yes	–	Unknown

* Queries cannot be drawn by hand, but are generated through a visual interface to the ontology.

An approach aimed at the graphical representation of RDF queries, developed by the Digital Enterprise Research Institute (DERI) at the National University of Ireland, is presented in [11]. The DERI graphical query language for RDF is built around facets - filter conditions over RDF graphs. The developed graphical language addresses however only a limited set of RDF queries.

Figure 1 presents a simple query, consisting of two facets, as enabled by the RDF graphical query language introduced in [11]. The purpose of this query consists of retrieving resources that have the keyword “RDF” in their title and address the subject of “metadata models”. It should further be noted that the output of DERI RDF GQL queries consists of regular RDF triples, which may serve as input to other queries, thus providing closure for the proposed graphical query language.

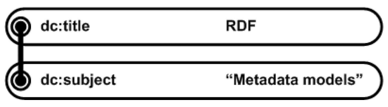


Fig. 1 RDF GQL graphical query example [11].

Although the language presented in [11] does not provide semantics for the introduced graphical constructs, the graphical queries may always be translated to N3 [2] query syntax. For the example query depicted in Fig. 1, the translation is shown in Fig. 2.

```
<> q1:select {
        ?subject1 ?p ?o .
}; q1:where {
        ?subject1 dc:title ?keyword .
        ?keyword yars:keyword ‘‘RDF’’ .
        ?subject1 dc:subject ‘‘Metadata models’’ .
        ?subject1 ?p ?o .
} .
```

Fig. 2 The example query in N3 query syntax.

XML-GL [8] is a graphical language for querying XML documents. The Graphical Data Model (GDM) introduced by the language addresses objects, properties, and relationships, represented as rectangles, circles, and arcs, respectively. Based hereon, XML-GL queries are then defined as consisting of four parts: i) the extract part, ii) the match part, iii) the clip part, iv) the construct part. Upon identifying the scope of the query in the *extract* part, the *optional* match part aims at representing additional logical conditions that should be imposed on the result set. The *clip* part specifies the focus of the query (relating to entities) in a similar way the *select* clause is used in SQL queries. Finally, the optional *construct* part of an XML-GL query specifies the new elements to be included in the result document and their relationships to the extracted elements [8].

Figure 3 depicts a graphical representation of an XML-GL query. The aim of this query is to select all (CD) items for which the product of the price and the quantity is less than 50. As can be observed from this figure, arithmetical operators may also be employed in the language, for the construction of complex queries such as the one presented here.

Despite fulfilling most of the requirements defined in [8], the language still lacks a precise definition of the semantics for the graphical symbols. Relevant to the current context, XML-GL is designed for XML, rather for more expressive knowledge representation languages such as RDF or OWL.

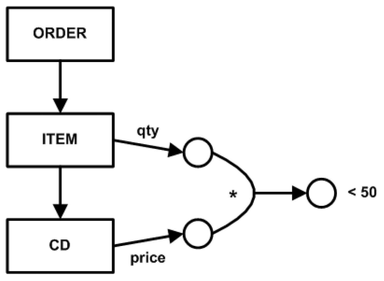


Fig. 3 XML-GLgraphical query example [8].

A graphical query language for OWL-DL ontologies (GLOO) is presented in [9]. The main focus of GLOO consists of translating visual, diagrammatic queries to DL-based query languages. The proposed version maps the graphical queries to the new Racer Query Language (nRQL) [10], but without matching the full expressive power of the latter language.

GLOO allows for construction of queries based on classes, individuals, and roles. Additionally, a number of operators may be employed: negation (true classical negation), complement (negation as failure), disjunction and conjunction. An example GLOO query is depicted in Fig. 4. The aim of this query is in selecting those sentences which have a human as subject and object (both conditions must be simultaneously satisfied), where *sentence* is a variable.

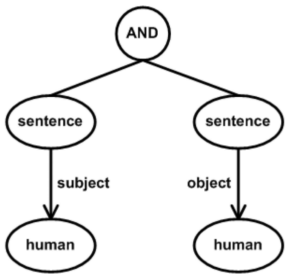


Fig. 4 GLOO example query [9].

The authors argue for the formality of the proposed language, employing as main argument the connectivity syntax on which GLOO is based, and that is defined based on a formal grammar [9]. Additionally, the way in which elements of a query are placed into space has no influence on the semantics of that query.

The EROS tool [19], is aimed at simplifying queries on RDFS models. The main focus of the tool is to combine the advantages of a tree-based approach and a graph-based approach for visualizing RDFS, as both approaches present advantages and shortcomings. A tree view has the downside of being somewhat limited, due to the fact that multiple inheritance is not visible, while a graph view is limited because of its hierarchical structure that is hard to discover. On the positive side, a tree view provides increased clarity when visualizing the relevant entities. However, one can express more complex structures with graphs than is possible by using trees. The combination of the two approaches has resulted in an interface with two hierarchy trees: one domain tree and one range tree. Properties are depicted as arrows from left to right between tree nodes. The property-centric view of EROS is in line with the RDF philosophy.

Despite the fact that EROS is developed for visualizing ontologies, it also offers a built-in query generator [19]. This generator is based on RQL [13] - a query language for RDF descriptions. RQL uses SELECT, FROM and WHERE clauses. The EROS user is able to generate queries by selecting nodes in the graph and assigning properties to them using normal buttons, listboxes, etc. in the interface. The user can specify which variables should be visible in the results.

Summarizing, EROS does not implement a graphical query language, but visualizes ontologies and enables the user to query RDFS models using a normal graphical interface. Vdovjak et al. claim that an effective visual representation of ontologies is vital for users, since querying models without a clear view of the ontology is cumbersome [19]. EROS provides an interface in which the user is able to view the ontology both from the viewpoint of classes and that of properties.

SPARQLViz [4] is a query editor centered around graphical query composition and natural language processing in an RDF visualization interface. This tool is an extension for IsaViz, a visual interaction tool for RDF. SPARQLViz implements graphical query composition by using a graphical user interface for generating SPARQL queries. The user has to click through different menus to compose a query, as presented in screenshots in [4]. The tool demonstrates that it is possible to cover a great part of the SPARQL syntax with a simple user interface. However, no graphical query language is implemented, which makes the understanding of the relationships between different query parts difficult.

In [1], GRQL is introduced. GRQL is an intuitive interface which is able to construct RQL queries (like EROS) by using inputs from the user via a graphical interface (for screenshots, see [1]). GRQL is a graphical query generator in a way that it uses a graphical user interface. GRQL does not implement a graphical query language and thus does not support drawing queries. With GRQL, the user is able to browse through an RDFS model and to generate a lot of different queries graphically. GRQL's tree-based interface offers many functionalities. It is able to handle a lot of different actions, varying from browsing RDFS models towards all directions and the possibility to translate a sequence of browsing actions into an RQL query.

The SEWASIE project (which stands for SEmantic Webs and AgentS in Integrated Economies) [6] shows us it is possible to create a tool with which a user can generate a query using an integrated ontology. The tool's main purpose is to offer functionality to generate conjunctive queries ready to be executed by some evaluation engine associated to the information system. With the SEWASIE tool, a user is able to compose a query using drop-down menus and input fields (shown in screenshots in [6]). The composed query can be viewed in a natural language-like form and in a graphical form. The authors, however, remain unclear on how query execution works with the application, and only demonstrate the capabilities of the query editor.

One of the main conclusion supported by the approaches presented in this section is that, currently, no graphical query language based on SPARQL exists for RDF. In the following section we introduce one such language in the form of RDF-GL, aimed at querying RDF ontologies through a translation of graphical queries to the state-of-the-art RDF query language - SPARQL.

3 RDF-GL

In this section we introduce RDF-GL, the first SPARQL-based graphical language for RDF. The main constructs of the language are presented in Sect. 3.1, whereas Sect. 3.2 elaborates on the subset of SPARQL which is covered by RDF-GL. Sections 3.3 and 3.4 present how SPARQL queries are mapped to RDF-GL, and how the latter can be converted into SPARQL queries, respectively.

3.1 Language Constructs

The constructs of RDF-GL, which shall be denoted as *elements*, can be divided into three main groups: boxes, circles and arrows. All elements can be found in the example query presented in Figs. 5 and 6. The elements of RDF-GL queries are assigned meaning based on their shape and color. In what follows, we provide an informal overview hereof.

Boxes can have an orange, pink or green color, each color representing different SPARQL query elements. An orange box, which is called a result box or simply *BR*, contains information about the execution of the query, e.g., the type of query and the way result variables are ordered. A pink box (referred to as an subject/object box or as *BSO*) represents a subject or an object of a triple in a SPARQL query, whereas a green box, the filtered subject/object box (*BFSO*), is used to depict filtered subjects or objects.

We do have two types of circles. Blue circles, called union circles (*CU*), are used in an RDF-GL query to define or-relationships, similar to SPARQL

```
PREFIX j.1: <http://www.daml.org/2003/09/factbook/factbook-ont#>
SELECT DISTINCT ?name ?oil
WHERE
{
    ?country j.1:localShortCountryName ?name .
    ?country j.1:grossDomesticProductPerCapita ?gdp .
    {
        FILTER (?gdp < 1500) .
    }
    UNION
    {
        FILTER (?gdp > 2500) .
    }
    OPTIONAL
    {
        ?country j.1:oilProvedReserves ?oil .
    }
}
ORDER BY ASC(?gdp)
```

Fig. 5 Example SPARQL query.

UNION blocks. Purple circles – to which we refer to as optional circles (CO) – are used for RDF-GL's equivalents of SPARQL OPTIONAL blocks, in order to identify statements which are optional.

RDF-GL uses four colors for arrows. Black arrows, labeled with a property, depict a SPARQL triple predicate and are referred to as property arrows (AP), since they can be interpreted as a property relationship between two elements (a subject and an object), whereas grey arrows (also known as optional arrows, or AO) are used to indicate optional statements. Yellow and red arrows point to relationships belonging to a SPARQL UNION block, where yellow arrows point to the first part of the block and the red arrows to the second part. Two types of UNION arrows are used, because it is implied by SPARQL, as SPARQL makes a distinction between the two block parts

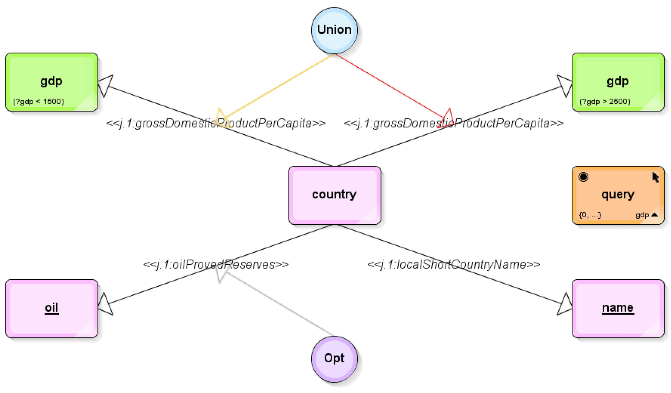

Union
gdp
(?gdp < 1500)
gdp
(?gdp > 2500)
<<j.1:grossDomesticProductPerCapita>>
<<j.1:grossDomesticProductPerCapita>>
country
query
{0, ...}
gdp
<<j.1:oilProvedReserves>>
<<j.1:localShortCountryName>>
oil
name
Opt

Fig. 6 Example RDF-GL query.

to be joined. Yellow and red arrows are also called union arrows (*AU1* and *AU2*).

The use of the different elements of an RDF-GL query can best be illustrated by means of an example. Let us assume we want to search the CIA World Factbook [7] for countries that have a gross domestic product per capita of less than $1,500 or greater than $2,500. We want to know the name of every country matching this criterion. Furthermore we want to know the oil supply of every resulting country, if any data about oil supply is stored in the database for these countries. The SPARQL query used for retrieving this data is presented in Fig. 5, whereas the RDF-GL graphical representation of this query is presented in Fig. 6. Note that from now on, we will refer to the ontology as *j.1* in our RDF-GL graphs to maintain readability. This prefix is declared in the SPARQL query and does not need to be declared explicitly in RDF-GL. In our RDF-GL, the prefix is thus considered given, as well as the prefixes RDF, RDFS, and XSD.

We next move on to a more formal presentation of the different elements of RDF-GL queries. In general, we denote an RDF-GL query by Q. Equations 1 through 4 give an overview of all possible elements of Q, and their different types:

$$Q = \{BOX,\ CRC,\ ARR\}\ , \tag{1}$$

$$BOX = \{BR,\ BSO,\ BFSO\}\ , \tag{2}$$

$$CRC = \{CU,\ CO\}\ , \tag{3}$$

$$ARR = \{AP,\ AO,\ AU1,\ AU2\}\ . \tag{4}$$

The different types of boxes (*BR*, *BSO*, and *BFSO*) are grouped in the *BOX* set. The circles joined in set *CRC* are referred to as *CU* and *CO*, which are – as stated earlier – the blue and purple circle, respectively. Finally, the black, grey, yellow and red arrows (*AP*, *AO*, *AU1*, and *AU2*, respectively) are stored in set *ARR*.

Tables 2 and 3 give an overview of the constructs of RDF-GL introduced in this section. For each construct we give its shape name, acronym, and color, as well as a short description.

Table 2 Constructs of RDF-GL: shapes, properties and names.

Subset	Name	Acronym	Color
Box (*BOX*)	Result box	*BR*	Orange
	Subject/object box	*BSO*	Pink
	Filtered subject/object box	*BFSO*	Green
Circle (*CRC*)	Union circle	*CU*	Blue
	Optional circle	*CO*	Purple
Arrow (*ARR*)	Property arrow	*AP*	Black
	Optional arrow	*AO*	Grey
	Union arrow 1	*AU1*	Yellow
	Union arrow 2	*AU2*	Red

Table 3 Constructs of RDF-GL: descriptions.

Element	Description
BR	Contains information about query execution
BSO	Subject or object in a SPARQL triple
BFSO	Filtered subject or object in a SPARQL triple
CU	SPARQL UNION block
CO	SPARQL OPTIONAL block
AP	Predicate in a SPARQL triple
AO	Points to an optional element
AU1	Points to an alternative element (part 1)
AU2	Points to an alternative element (part 2)

In what follows, we focus on giving a more precise description for each type of element of an RDF-GL query, given in Fig. 7. We start with describing drawing rules for boxes and continue by elaborating on circles. Finally, arrows are discussed.

3.1.1 Boxes

In general, box shapes have five positions where properties can be defined. There is one position in each corner, and one position in the center of the figure. Figure 7(a) shows a basic box shape. The positions are indicated with B_1 to B_5. The different types of boxes in our graphical query language not only differ in color, but also in the positions they use and how they use them. We continue with describing each type of box separately.

The *BR* box contains information about the execution of the query. In an RDF-GL query, exactly one *BR* box should be present. Also, its properties are bound to rules and restrictions. The box can be neither a child of another element in the query (i.e., being on the receiving end of a property relationship) nor parent of another element in the query (i.e., having a property relationship). Additionally, some graphic rules apply.

First of all, the *BR* box should be orange. Furthermore, a query name should be depicted, which is the centered text on position B_5 in the example. Subsequently, the corners of the box may each contain information.

The upper-left corner (B_1) states whether the query should return only distinct values or not. The upper-right corner (B_2) contains information about the SPARQL query type, which can be only SELECT for the moment, and the lower-right corner (B_3) is reserved for ordering the results. The names of

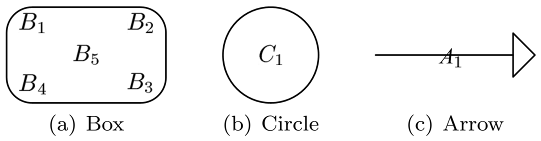


Fig. 7 Basic shapes of RDF-GL.

the variables by which the query results should be ordered (zero or more) are displayed in this corner, where each name is followed by a symbol indicating an ascending or descending ordering. The lower-left corner (B_4) contains information about the result range, displayed as $\{$*from, to*$\}$. For example, if the range is set to $\{$*5, 8*$\}$, results 5, 6, 7 and 8 will be displayed as results. It should be noted that ending the range with "*...*" indicates infinity (for result set length). Table 4 shows all *BR* symbols with their descriptions.

Table 4 Overview of symbols used in a *BR* box in RDF-GL.

Corner	Symbol	Explanation
Upper-left	○	Display all results
	◉	Display only distinct results
Upper-right	↖	SELECT query
Lower-right	▲	Ascending ordering
	▼	Descending ordering
Lower-left	{ , }	Range of results

The other boxes do not use all positions. The *BSO* box is a pink rectangle, which represents a subject or object in a SPARQL triple. In RDF-GL, the subject and object types are limited to a (new) variable, a blank node, an ontology object, or data type. A variable is displayed as bold text on position B_5 (representing the variable name), with or without underline. By underlining the variable name, one can express the variable will be visible in the query results. In case the box lacks a name, it represents a blank SPARQL node. Finally, one can denote an object or data type as a *BSO* box by placing its type on position B_5, for example ≪*type*≫. A *BSO* box can be a child of another *BSO* or *BFSO* box and can also be parent of another *BSO* or *BFSO* box.

The same rules apply to the *BFSO* box, which represents a filtered subject or object, except for the color, which is green instead of pink. Also, *BFSO* boxes cannot have empty names (i.e., cannot represent a blank SPARQL node) and their types are restricted to (new) variables. The content of the applied filter is displayed on position B_4, which is the lower-left corner of the rectangle shaped construct.

3.1.2 Circles

Circles only have one position which can be given a property. This position, C_1 (shown in Fig. 7(b)), should always be used and is located in the shape's center.

A *CU* circle is used for representation of a SPARQL UNION block, which models alternatives. In the center of this blue circle, "Union" is depicted. Restrictions of the *CU* are that it can only be a child and/or parent of both elements from the *CRC* set. Also, a *CU* can be parent of an *AP* arrow.

Finally, *AU1* and *AU2* are associated with this circle. Note that conjunctions are implicit in SPARQL and thus are not included in RDF-GL as a *CU*-like symbol.

The purple colored *CO* circles, which are all labeled with "Opt", represent a SPARQL OPTIONAL block. These circles can be a child and/or a parent of another circle in *CRC*. Also, *CO* circles can be parent of an *AP* arrow. Grey arrows, *AO*, are used in combination with the *CO* circle. Note that RDF-GL elements which are connected by arrows have parent-child relationships. Children are on the receiving end of an arrow, while parents are on the other side of an arrow. Using circles, one can create nested OPTIONAL and UNION blocks, simply by pointing an *AO*, *AU1*, or *AU2* arrow from one circle to another circle. If an arrow points from a circle to an arrow, it represents a SPARQL triple inside an OPTIONAL or UNION block.

3.1.3 Arrows

Figure 7(c) shows the basic shape of an arrow. Each arrow is constructed with a transparent, closed head. As is the case with circles, arrows have only one property position in their center: A_1.

Four types of arrows can be distinguished, which are included in the *ARR* set. The black arrow, *AP*, should be read as a property relationship between two elements, for example: a resulting country has a gross domestic product per capita of less than $2,500. The arrow represents a SPARQL triple predicate. Property types can be object or data types from ontologies, variables previously defined in the query and new variables. The property type is specified as a label located in the center of the arrow (position A_1). An *AP* arrow can be drawn from and to *BSO* and *BFSO* boxes.

An *AO* arrow indicates a SPARQL OPTIONAL relationship between two elements. Just like the *AP* arrow, the *AO* arrow should be read as a property relationship between two elements, but an *AO* arrow can only be drawn between a *CO* circle and the two types of circles or between a *CO* circle and an *AP* arrow.

As mentioned earlier, *AU1* and *AU2* arrows can be used in combination with the *CU* circles. With *AU1* arrows, one is able to define which relationships belong to the first part of the SPARQL UNION block and with *AU2* arrows one can define which belongs to the second part of the SPARQL UNION block. These two arrows can only be drawn from a *CU* circle to both circle types or from a *CU* circle to a black arrow.

An important point relates to the fact that arrows cannot be drawn from and to every element in a query. Also, it is not possible for an element to have children of every type. Tables 5 and 6 summarize the ways in which various RDF-GL elements may be connected.

Table 5 shows the allowed directions for every arrow type (displayed as columns) with respect to every element type (displayed as rows). Allowed directions are: from an element (from), to an element (to), and none (–).

As we can see, no arrows can be drawn from or to *BR* boxes. In RDF-GL, one is allowed to draw *AP* arrows from and to *BSO* and *BFSO* boxes. Furthermore, *AO* arrows cannot be drawn from or to other *AO*, *AU1*, or *AU2* arrows, as well as from or to other boxes. However, these arrows can be drawn to both types of circles and *AP* arrows, and can also be drawn from *CO* circles. The *AU1* and *AU2* arrows have equal restrictions to those of the *AO* arrows. However, the shapes differ in that *AO* arrows may only be drawn from *CO* circles, whereas *AU1* and *AU2* arrows may only be drawn from *CU* circles.

Table 5 Overview allowed arrow directions RDF-GL.

	Arrow			
Element	*AP*	*AO*	*AU1*	*AU2*
BR	–	–	–	–
BSO	from/to	–	–	–
BFSO	from/to	–	–	–
CU	–	to	from/to	from/to
CO	–	from/to	to	to
AP	–	to	to	to
AO	–	–	–	–
AU1	–	–	–	–
AU2	–	–	–	–

Since arrows indicate parent-child relationships, we can deduce Table 6 from Table 5. Table 6 shows every possible parent-child relationship. Possible parents are all types of boxes and circles, which are displayed in the columns of the table. Possible children are all elements of an RDF-GL query, which are displayed in the rows of the table. In Table 6 we summarize the valid parent-child relationships, where (+) denotes a valid relationship and (–) an invalid (not allowed) one.

As can be observed from this table, the orange box cannot be parent of any element in an RDF-GL query. The *BSO* and *BFSO* boxes can only be

Table 6 Overview allowed parent-child relationships RDF-GL.

	Parent								
Child	*BR*	*BSO*	*BFSO*	*CU*	*CO*	*AP*	*AO*	*AU1*	*AU2*
BR	–	–	–	–	–	–	–	–	–
BSO	–	+	+	–	–	–	–	–	–
BFSO	–	+	+	–	–	–	–	–	–
CU	–	–	–	+	+	–	–	–	–
CO	–	–	–	+	+	–	–	–	–
AP	–	–	–	+	+	–	–	–	–
AO	–	–	–	–	–	–	–	–	–
AU1	–	–	–	–	–	–	–	–	–
AU2	–	–	–	–	–	–	–	–	–

parent of other *BSO* and *BFSO* boxes. Furthermore, both types of circles can be parent of *AP* arrows and both types of circles. Finally, *AO*, *AU1* and *AU2* arrows cannot have any parents and none of the arrows can have children.

3.2 The SPARQL Subset of RDF-GL

In what follows, we define the subset of SPARQL which can be covered using the elements of the RDF-GL query language by means of Extended BackusNaur Form (EBNF) [15] rules, which are similar to the ones defined for SPARQL in [14]. Most of the rules in [14] can be maintained. However, since RDF-GL only covers a subset of SPARQL, we need to alter some of the grammar rules in order to define the covered SPARQL subset adequately.

First of all, SPARQL queries can be either ASK, CONSTRUCT, DESCRIBE, or SELECT queries. Usually, a query can be defined as a prologue, followed by a query type. This prologue contains BASE and PREFIX statements. In RDF-GL, currently only SELECT queries are covered, and thus we can define our first rule, which differs from SPARQL in that the prologue and all query types but the SELECT query are removed.

```
Query ::= SelectQuery
```

Normally, in SPARQL, a SELECT query consists of the string “SELECT”, optionally followed by the string “DISTINCT” or “REDUCED”, followed by one or more variables which have to be selected to be returned in the result set, zero or more data set clauses containing FROM and NAMED elements, a WHERE clause, and solution modifiers. The subset of SPARQL which covers RDF-GL does not include all the elements of a regular SPARQL SELECT query [14]. The string “REDUCED” is not supported, as well as the data set clause (FROM and NAMED). Therefore, our second rule also differs from the rule presented in the SPARQL grammar.

```
SelectQuery ::= 'SELECT' 'DISTINCT'? ( Var+ | '*' ) WhereClause SolutionModifier
```

SPARQL implements two types of variables, which have a name preceded by either a “?” or a “$” (type 1 and 2, respectively). RDF-GL can currently only represent the former type, and thus we define another rule which differs from the one presented in the SPARQL grammar.

```
Var ::= VAR1
```

Continuing defining the grammar rules of the SPARQL subset, we can state that the WHERE clause is not fully supported by RDF-GL. Normally, this clause would contain triples, FILTER elements, and graph patterns which are not triples, i.e., OPTIONAL, UNION, and GRAPH elements. RDF-GL’s SPARQL subset does not contain GRAPH elements, but the triples and FILTER elements as defined in the clause are fully included. Therefore, we can add three rules to our rule set. The first two rules are exactly the same as in the grammar of SPARQL, but the last rule is redefined so that it cannot contain GRAPH elements. The rules are the following.

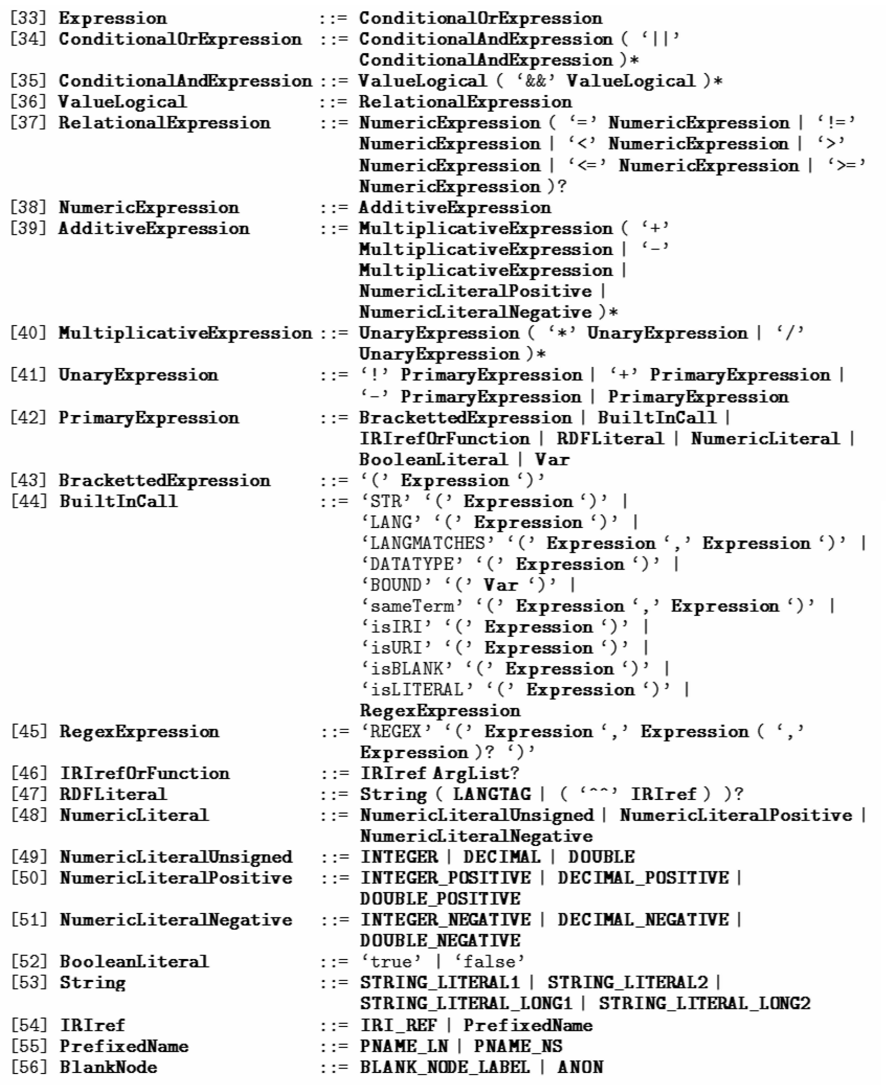
```
WhereClause            ::= 'WHERE'? GroupGraphPattern
GroupGraphPattern      ::= '{' TriplesBlock? ( ( GraphPatternNotTriples |
                           Filter ) '.'? TriplesBlock? )* '}'
GraphPatternNotTriples ::= OptionalGraphPattern | GroupOrUnionGraphPattern
```

Furthermore, the solution modifiers of the SELECT query (i.e., ORDER BY, LIMIT, and OFFSET) belong to the subset of SPARQL that can be represented by elements of RDF-GL.

When all non-terminal rules are refined using the rules of the SPARQL grammar, which of course all apply to some extent to our subset, the obtained rule set is as given in Fig. 8. Figure 9 shows all terminals.

```
[1]  Query                    ::= SelectQuery
[2]  SelectQuery              ::= 'SELECT' 'DISTINCT'? ( Var+ | '*' ) WhereClause
                                  SolutionModifier
[3]  WhereClause              ::= 'WHERE'? GroupGraphPattern
[4]  SolutionModifier         ::= OrderClause? LimitOffsetClauses?
[5]  LimitOffsetClauses       ::= ( LimitClause OffsetClause? | OffsetClause
                                  LimitClause? )
[6]  OrderClause              ::= 'ORDER' 'BY' OrderCondition+
[7]  OrderCondition           ::= ( ( 'ASC' | 'DESC' ) BrackettedExpression )
                                  ( Constraint | Var )
[8]  LimitClause              ::= 'LIMIT' INTEGER
[9]  OffsetClause             ::= 'OFFSET' INTEGER
[10] GroupGraphPattern        ::= '{' TriplesBlock? ( ( GraphPatternNotTriples |
                                  Filter ) '.'? TriplesBlock? )* '}'
[11] TriplesBlock             ::= TriplesSameSubject ( '.' TriplesBlock? )?
[12] GraphPatternNotTriples   ::= OptionalGraphPattern | GroupOrUnionGraphPattern
[13] OptionalGraphPattern     ::= 'OPTIONAL' GroupGraphPattern
[14] GroupOrUnionGraphPattern ::= GroupGraphPattern ( 'UNION' GroupGraphPattern )*
[15] Filter                   ::= 'FILTER' Constraint
[16] Constraint               ::= BrackettedExpression | BuiltInCall | FunctionCall
[17] FunctionCall             ::= IRIref ArgList
[18] ArgList                  ::= ( NIL | '(' Expression ( ',' Expression )* ')' )
[19] TriplesSameSubject       ::= VarOrTerm PropertyListNotEmpty | TriplesNode
                                  PropertyList
[20] PropertyListNotEmpty     ::= Verb ObjectList ( ';' ( Verb ObjectList )? )
[21] PropertyList             ::= PropertyListNotEmpty?
[22] ObjectList               ::= Object ( ',' Object )*
[23] Object                   ::= GraphNode
[24] Verb                     ::= VarOrIRIref | 'a'
[25] TriplesNode              ::= Collection | BlankNodePropertyList
[26] BlankNodePropertyList    ::= '[' PropertyListNotEmpty ']'
[27] Collection               ::= '(' GraphNode+ ')'
[28] GraphNode                ::= VarOrTerm | TriplesNode
[29] VarOrTerm                ::= Var | GraphTerm
[30] VarOrIRIref              ::= Var | IRIref
[31] Var                      ::= VAR1
[32] GraphTerm                ::= IRIref | RDFLiteral | NumericLiteral |
                                  BooleanLiteral | BlankNode | NIL
```

Fig. 8 Rules in RDF-GL's subset of SPARQL (non-terminals).



```
[33] Expression                ::= ConditionalOrExpression
[34] ConditionalOrExpression   ::= ConditionalAndExpression ( '||'
                                   ConditionalAndExpression )*
[35] ConditionalAndExpression ::= ValueLogical ( '&&' ValueLogical )*
[36] ValueLogical              ::= RelationalExpression
[37] RelationalExpression      ::= NumericExpression ( '=' NumericExpression | '!='
                                   NumericExpression | '<' NumericExpression | '>'
                                   NumericExpression | '<=' NumericExpression | '>='
                                   NumericExpression )?
[38] NumericExpression         ::= AdditiveExpression
[39] AdditiveExpression        ::= MultiplicativeExpression ( '+'
                                   MultiplicativeExpression | '-'
                                   MultiplicativeExpression |
                                   NumericLiteralPositive |
                                   NumericLiteralNegative )*
[40] MultiplicativeExpression ::= UnaryExpression ( '*' UnaryExpression | '/'
                                   UnaryExpression )*
[41] UnaryExpression           ::= '!' PrimaryExpression | '+' PrimaryExpression |
                                   '-' PrimaryExpression | PrimaryExpression
[42] PrimaryExpression         ::= BrackettedExpression | BuiltInCall |
                                   IRIrefOrFunction | RDFLiteral | NumericLiteral |
                                   BooleanLiteral | Var
[43] BrackettedExpression      ::= '(' Expression ')'
[44] BuiltInCall               ::= 'STR' '(' Expression ')' |
                                   'LANG' '(' Expression ')' |
                                   'LANGMATCHES' '(' Expression ',' Expression ')' |
                                   'DATATYPE' '(' Expression ')' |
                                   'BOUND' '(' Var ')' |
                                   'sameTerm' '(' Expression ',' Expression ')' |
                                   'isIRI' '(' Expression ')' |
                                   'isURI' '(' Expression ')' |
                                   'isBLANK' '(' Expression ')' |
                                   'isLITERAL' '(' Expression ')' |
                                   RegexExpression
[45] RegexExpression           ::= 'REGEX' '(' Expression ',' Expression ( ','
                                   Expression )? ')'
[46] IRIrefOrFunction          ::= IRIref ArgList?
[47] RDFLiteral                ::= String ( LANGTAG | ( '^^' IRIref ) )?
[48] NumericLiteral            ::= NumericLiteralUnsigned | NumericLiteralPositive |
                                   NumericLiteralNegative
[49] NumericLiteralUnsigned    ::= INTEGER | DECIMAL | DOUBLE
[50] NumericLiteralPositive    ::= INTEGER_POSITIVE | DECIMAL_POSITIVE |
                                   DOUBLE_POSITIVE
[51] NumericLiteralNegative    ::= INTEGER_NEGATIVE | DECIMAL_NEGATIVE |
                                   DOUBLE_NEGATIVE
[52] BooleanLiteral            ::= 'true' | 'false'
[53] String                    ::= STRING_LITERAL1 | STRING_LITERAL2 |
                                   STRING_LITERAL_LONG1 | STRING_LITERAL_LONG2
[54] IRIref                    ::= IRI_REF | PrefixedName
[55] PrefixedName              ::= PNAME_LN | PNAME_NS
[56] BlankNode                 ::= BLANK_NODE_LABEL | ANON
```

Fig. 8 Rules in RDF-GL's subset of SPARQL (non-terminals), continued.

3.3 Mapping SPARQL to RDF-GL

This section explains how the most common features of a SPARQL SELECT query look like in RDF-GL, or in other words, how SPARQL is mapped to RDF-GL. We try to create a mapping using the main rules of Fig. 8, as discussed in Sect. 3.2.

```
[57] IRI_REF              ::= ‘<’ ( [^<>"{}|^‘\]-[#x00-#x20] )* ‘>’
[58] PNAME_NS             ::= PN_PREFIX? ‘:’
[59] PNAME_LN             ::= PNAME_NS PN_LOCAL
[60] BLANK_NODE_LABEL     ::= ‘_:’ PN_LOCAL
[61] VAR1                 ::= ‘?’ VARNAME
[62] LANGTAG              ::= ‘@’ [a-zA-Z]+ ( ‘-’ [a-zA-Z0-9]+ )*
[63] INTEGER              ::= [0-9]+
[64] DECIMAL              ::= [0-9]+ ‘.’ [0-9]* | ‘.’ [0-9]+
[65] DOUBLE               ::= [0-9]+ ‘.’ [0-9]* EXPONENT | ‘.’ ( [0-9] )+
                              EXPONENT | ( [0-9] )+ EXPONENT
[66] INTEGER_POSITIVE     ::= ‘+’ INTEGER
[67] DECIMAL_POSITIVE     ::= ‘+’ DECIMAL
[68] DOUBLE_POSITIVE      ::= ‘+’ DOUBLE
[69] INTEGER_NEGATIVE     ::= ‘-’ INTEGER
[70] DECIMAL_NEGATIVE     ::= ‘-’ DECIMAL
[71] DOUBLE_NEGATIVE      ::= ‘-’ DOUBLE
[72] EXPONENT             ::= [eE] [+-]? [0-9]+
[73] STRING_LITERAL1      ::= "‘" ( ( [^#x27#x5C#xA#xD] ) | ECHAR )* "’"
[74] STRING_LITERAL2      ::= ‘"’ ( ( [^#x22#x5C#xA#xD] ) | ECHAR )* ‘"’
[75] STRING_LITERAL_LONG1 ::= "’’’" ( ( "’" | "’’" )? ( [^’\] | ECHAR ) )* "’’’"
[76] STRING_LITERAL_LONG2 ::= ‘"""’ ( ( ‘"’ | ‘""’ )? ( [^"\] | ECHAR ) )* ‘"""’
[77] ECHAR                ::= ‘\’ [tbnrf\"’]
[78] NIL                  ::= ‘(’ WS* ‘)’
[79] WS                   ::= #x20 | #x9 | #xD | #xA
[80] ANON                 ::= ‘[’ WS* ‘]’
[81] PN_CHARS_BASE        ::= [A-Z] | [a-z] | [#x00C0-#x00D6] |
                              [#x00D8-#x00F6] | [#x00F8-#x02FF] |
                              [#x0370-#x037D] | [#x037F-#x1FFF] |
                              [#x200C-#x200D] | [#x2070-#x218F] |
                              [#x2C00-#x2FEF] | [#x3001-#xD7FF] |
                              [#xF900-#xFDCF] | [#xFDF0-#xFFFD] |
                              [#x10000-#xEFFFF]
[82] PN_CHARS_U           ::= PN_CHARS_BASE | ‘_’
[83] VARNAME              ::= ( PN_CHARS_U | [0-9] ) ( PN_CHARS_U | [0-9] |
                              #x00B7 | [#x0300-#x036F] | [#x203F-#x2040] )*
[84] PN_CHARS             ::= PN_CHARS_U | ‘-’ | [0-9] | #x00B7 |
                              [#x0300-#x036F] | [#x203F-#x2040]
[85] PN_PREFIX            ::= PN_CHARS_BASE ( ( PN_CHARS | ‘.’ )* PN_CHARS )?
[86] PN_LOCAL             ::= ( PN_CHARS_U | [0-9] ) ( ( PN_CHARS | ‘.’ )*
                              PN_CHARS )?
```

Fig. 9 Rules in RDF-GL's subset of SPARQL (terminals).

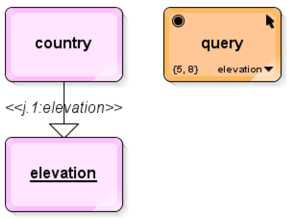
3.3.1 Query Type and Sequence Modifiers

As stated in Sect. 3.2, RDF-GL uses a subset of SPARQL, which results in the fact that only SELECT queries can be performed to a certain extent. The main elements of this query which are implemented in the RDF-GL language, are sequence modifiers, variables to include in the result set, and a WHERE clause (Rules 1 and 2). These sequence modifiers, i.e., DISTINCT, LIMIT, OFFSET, and ORDER BY (Rules 4, 5, 6, 8, 9), all can be defined using a *BR* box and the symbols from Table 4. Furthermore, the variables that have to be selected are denoted as pink or green (filtered) boxes with an underlined, centered label.

Figure 10 shows the translation from a SPARQL SELECT query that uses all sequence modifiers to an RDF-GL query. The displayed query asks for all distinct elevations of countries in the CIA World Factbook. Results 5, 6, 7 and 8 are returned in a descending order. The SPARQL triple (*?country j.1:elevation ?elevation .*) is drawn using two *BSO* boxes, both representing variables. Solely information on elevation will be returned in the result set, which is denoted by the underlining of the variable name in the RDF-GL query. In the *BR* box, all corners have been used to display the sequence modifiers.

```
PREFIX j.1: <http://www.daml.org/2003/09/factbook/factbook-ont#>
SELECT DISTINCT ?elevation
WHERE
{
    ?country j.1:elevation ?elevation .
}
ORDER BY DESC(?elevation) OFFSET 5 LIMIT 4
```

(a) SPARQL



(b) RDF-GL

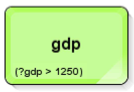
Fig. 10 Mapping query type and sequence modifiers.

3.3.2 Filtered Variables

One element that is included in the WHERE clause (Rules 3 and 10) of a SPARQL query in general, as well as in our implemented subset of SPARQL, is the FILTER element (Rule 15). RDF-GL has full functionality with respect

```
FILTER (?gdp > 1250) .
```

(a) SPARQL



(b) RDF-GL

Fig. 11 Mapping filters.

to filtering, as the SPARQL filter condition is embedded in the graphical representation.

In RDF-GL, filtering variables used in a query can be done by denoting the filtered variable as a *BFSO* box with a variable name and filter. This box is equal to a FILTER statement in a query written in SPARQL syntax. Figure 11 shows how a filter is applied to a variable called *gdp* in SPARQL and how the same filter can be applied to a variable in a RDF-GL query. In RDF-GL, the filter is displayed in the lower-left corner of the box representing the variable and the box has been colored green.

3.3.3 Triples

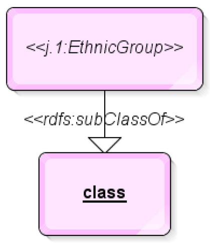
Another important element in the WHERE clause is the triple. Each query contains one or more triples. According to Rule 11 and 19 and their refinements, a triple typically consists of a variable or term, followed by a property and another variable or reference to an IRI from an ontology. A term can either be a reference to an IRI from an ontology, some literals (data types), or a blank node. A property is defined as a variable, IRI, or data type from an ontology (Rule 20 and some of the rules after it). The three elements of a triple are also called subject, predicate and object.

With RDF-GL, these elements are denoted as two *BSO* or *BFSO* boxes (representing the subject and object) and an *AP* arrow between them (representing the predicate). The arrow points from the box representing the triple's subject to the box representing the triple's object and is labeled with the predicate name. Both boxes and arrows are able to represent all required elements.

Figure 12 shows a single triple in SPARQL syntax and the same triple in RDF-GL. This triple asks for the classes of which the class *EthnicGroup* is a subclass and stores them in a variable called *class*. Two *BSO* boxes and one

```
j.1:EthnicGroup rdfs:subClassOf ?class .
```

(a) SPARQL



(b) RDF-GL

Fig. 12 Mapping triples.

arrow have been used to construct this triple in RDF-GL. The upper box represents the subject, and the lower box represents the object, which in this case is a variable. The arrow depicts the triple's predicate.

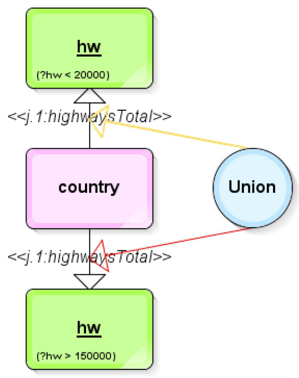
3.3.4 Alternatives

A third part of the WHERE clause of the full SPARQL set is not entirely covered by RDF-GL: the graph patterns which are not triples (Rule 12 shows what is covered). One of those patterns is called the UNION graph pattern (Rule 14), which is nothing more than an element which groups 2 query blocks (containing for example triples) and takes the union of both groups. This way one is able to represent alternatives.

In RDF-GL, a *CU* circle and *AU1* and *AU2* arrows are used to point to elements (triples and other graph patterns) which represent a union part. The user decides which triples belong to which part of the union, and draws the arrows accordingly. Whichever elements belong to the first group (at least 1) will be pointed at with a yellow arrow (*AU1*), and the other elements (also at least 1) will be pointed at with a red arrow (*AU2*). These arrows point from a *CU* circle to *AP* arrows (predicates of triples), other *CU* circles, or *CO*

```
?country j.1:highwaysTotal ?hw .
{
    FILTER (?hw < 20000) .
}
UNION
{
    FILTER (?hw > 150000) .
}
```

(a) SPARQL



(b) RDF-GL

Fig. 13 Mapping alternative triples.

circles. In case circles are being pointed at, it can lead to nested alternatives or options, which will be discussed shortly.

Figure 13 shows an alternative in SPARQL and how the same alternative is represented in RDF-GL. The UNION depicted in Fig. 13 joins a variable which is filtered in two different ways. The countries which have a total highway kilometers of less than 20,000 kilometers as well as the countries which have a total highway kilometers of more than 150,000 have to be selected.

3.3.5 Options

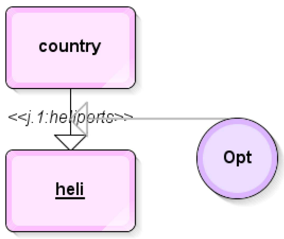
The rules we have defined for our SPARQL subset indicate that not only the UNION element is included in RDF-GL, but also the OPTIONAL element. In SPARQL, one is able to provide additional triples by using the OPTIONAL block (Rule 13).

In RDF-GL, a *CO* circle and *AO* arrows are used to point to triples which have to be marked as optional. The *AO* arrows point from *CO* circles to *AP* arrows (representing predicates of triples which have to be added to the set of additional triples).

Figure 14 shows an optional triple in SPARQL and how the same triple is marked as additional in our developed GQL. The triple depicted in Fig. 14 asks for the number of helicopter ports in a resulting country and stores it in a variable called *heli*.

```
OPTIONAL
{
    ?country j.1:heliports ?heli .
}
```

(a) SPARQL



(b) RDF-GL

Fig. 14 Mapping optional triples.

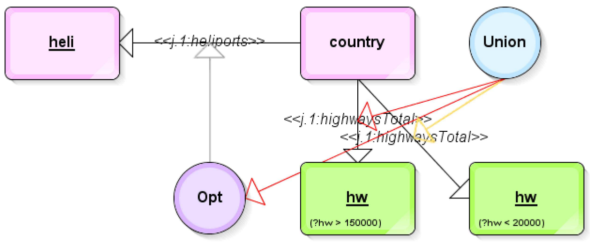
3.3.6 Nested Options and Alternatives

In SPARQL, it is possible to create nested options and alternatives. By looking at the defined grammar rules carefully, it becomes clear that UNION and OPTIONAL blocks not only include triples, but also other graph patterns which are not triples. Recalling our implementation of those graph patterns, we see that these patterns are in fact UNION and OPTIONAL blocks and thus it is possible to nest several options and alternatives in one query. This is also possible in RDF-GL. One can denote nested options and/or alternatives in RDF-GL by letting one or more *AO*, *AU1*, or *AU2* arrows point to circle(s). These arrows are not only allowed to point to *AP* arrows, but also to both types of circles.

Figure 15 combines the queries from Figs. 13 and 14 by nesting the option from Fig. 13 in the second union part of the query from Fig. 14. The order in which the union parts are specified is not relevant. Arrows pointing from the *CU* circle to the *AP* arrows indicate the triples to which these arrows belong should be regarded as alternatives. The circle to which an *AU2* arrow is pointing should also be added to the same alternative. The *AO* arrow pointing to an *AP* arrow indicates that the triple to which this arrow belongs should be optional.

```
?country j.1:highwaysTotal ?hw .
{
    FILTER (?hw < 20000) .
}
UNION
{
    FILTER (?hw > 150000) .
    OPTIONAL
    {
        ?country j.1:heliports ?heli .
    }
}
```

(a) SPARQL



(b) RDF-GL

Fig. 15 Mapping nested triples.

3.4 Converting RDF-GL to SPARQL

RDF-GL queries can be converted to SPARQL queries, using the algorithms presented in Figs. 16 and 17. These algorithms generate SPARQL queries based on drawing order. The SPARQL query is generated in a fixed order. First the default prefixes for RDF, RDFS, and XSD are generated, as well as for the ontology currently used (1). Subsequently, the query type is determined (2), after which the complete WHERE clause is generated (3). Finally, the ORDER BY (7) and the LIMIT and OFFSET (8) statements are determined. These main steps are directly related to some of the basic rules we defined in Sect. 3.2 (Rules 1 and 2, as well as 3 and 7).

Generating the prefixes and fetching the query type (with or without DISTINCT parameter and variables to select) is quite straightforward. Generating the complete WHERE clause, however, involves more complex actions. First, all *ARR* elements are read and converted to triples (4). We have seen in Rules 10, 11, and 19, that many types of triple configurations exist. If these triples do not belong to a UNION or OPTIONAL block, they are added to the SPARQL query (5). Subsequently, all *CU* circles and *CO* circles with their children (triples or other circles) are added to the query (6), using the recursive algorithm shown in Fig. 17. The generation of the ORDER BY, LIMIT and OFFSET is trivial and is solely based on the Rules 6 to 9.

4 SPARQLinG

This chapter introduces our RDF-GL editor: SPARQLinG. We elaborate on the technical details of this editor and provide an overview of the application's functionality. Finally, we present experimental results on the SPARQLinG tool.

4.1 Design

The SPARQLinG RDF-GL editor is a Java-based editor, which is able to read an RDF file (which contains both schema and instance data) and interpret the ontologies used, and offers users with little knowledge of SPARQL and some knowledge on the domain of the RDF file tools to draw RDF-GL queries in an intuitive way. Furthermore, RDF-GL queries can be converted into SPARQL queries and can be executed.

Although quite a few Java libraries for drawing graphs are around, such as JGraph [17], Piccolo [12], and Prefuse [18], none of these are suitable for SPARQLinG, since real-time drawing mostly is not supported and it is difficult to store non-standard information in the graph elements of the libraries.

```
Data: all elements from drawing
Result: RDF-GL converted to SPARQL
query = "";
query += prefixes;                                                        (1)
query += BR.type;                                                         (2)
if BR.type = SELECT then
    if BR.distinct = true then
        query += DISTINCT;
    foreach arrow in ARR and box in BOX do
        if show = true and type = variable and is not in SELECT then
            query += name;
    end
    query += WHERE;                                                       (3)
    foreach arrow in ARR do
        triple = "";                                                      (4)
        //Subject
        foreach box in BOX do
            if box.id = arrow.fromId then
                triple += box.name, blank or box.objectType;
                if filter present and box.type = variable then
                    store filter;
        end
        //Predicate
        triple += arrow.name or arrow.objectType;
        //Object
        foreach box in BOX do
            if box.id = arrow.toId then
                triple += box.name, blank or box.objectType;
                if filter present and box.type = variable then
                    store filter;
        end
        store triple with filter in triples;
    end
    foreach triple in triples do
        search for references in AU1, AU2, and AO;                        (5)
        if no references found then
            query += triple;
            if triple has filter then
                query += filter;
    end
    foreach circle in CU and CO do
        search for parentless circle;                                     (6)
        if found then
            query += getChildren(id);
    end
    query += BR.orderBy;                                                  (7)
    convert BR.range to limit and offset;                                 (8)
    query += limit;
    query += offset;
    return query;
```

Fig. 16 Generating a SPARQL query (generateQuery).

```
**Result**: triples and children's triples
**Input**: id of circle
query = "";
store all circle children which are triples in triples;
forall triple in triples do
    query += triple;
end
store all circle children which are other circles in CRC;
forall circle in CRC do
    query += getChildren(circle.id);
end
return query;
```

Fig. 17 Generating a SPARQL query (getChildren).

Also, these libraries cause a lot of overhead. Therefore, both the functionality of the graphical user interface and the graphics are created without using any existing libraries. Reading and interpreting RDF files however, is done using the Jena [16] library. The latter library is also used for executing SPARQL queries.

At an abstract level, we distinguish between three main components of SPARQLinG: i) Ontology Management, ii) Query Drawing, and iii) Query Execution. In what follows, we describe the main functionality hereof and discuss the interactions between components, which is also illustrated in Fig. 18.

An RDF file which has to be queried is fed into the *Ontology Management* component. The RDF Schema ontology used in the RDF file is extracted, so that it can be used in the *Query Drawing* component. Also, the RDF instances which populate the RDF Schema ontology are extracted. Both RDF instances and RDF Schema are used in the *Query Execution* component.

After loading an RDF file, the *Query Drawing* component offers the user tools to draw RDF-GL queries and handles all interface tasks. RDF-GL elements can be drawn using all elements stored in the ontology. The RDF-GL query is fed into the third module, the *Query Execution* module.

The *Query Execution* module handles two tasks. The first task is converting an RDF-GL query into a SPARQL query, which is done using the algorithms elaborated on in Sect. 3.4. The second task is executing this SPARQL

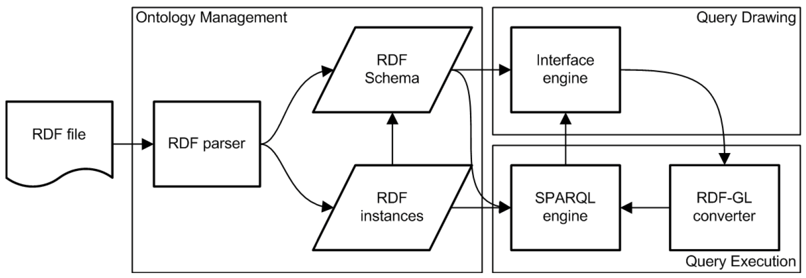
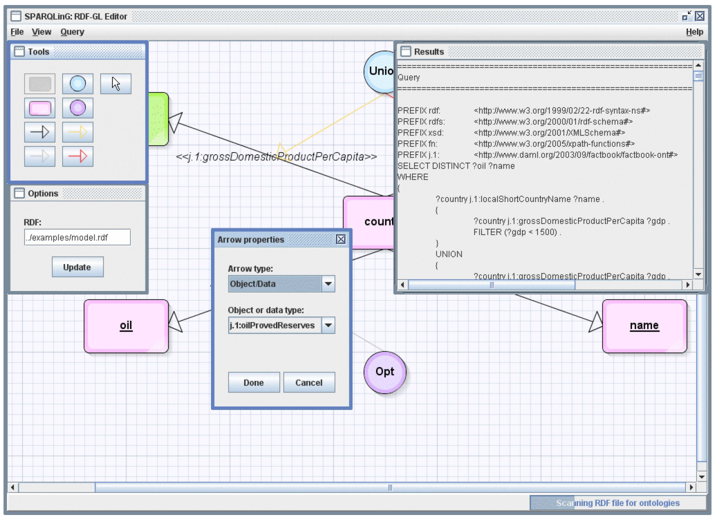


Fig. 18 Design of SPARQLinG.

query, using the ontology and RDF instances read from the input file. The query results are returned to the *Query Drawing* component.

4.2 Using SPARQLinG

The SPARQLinG RDF-GL editor is a tool like many other drawing applications. The user interface contains floating windows, which can be moved and toggled on and off. These windows contain drawing tools, settings, and query results. Furthermore, hot-keys are implemented for several actions, such as opening and saving files and executing queries. Figure 19 shows the user interface of SPARQLinG.



RDF: ./examples/model.rdf

Arrow type: Object/Data

Object or data type: j.1:oilProvedReserves

```
==========================================================================
Query
==========================================================================

PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX fn:       <http://www.w3.org/2005/xpath-functions#>
PREFIX j.1:      <http://www.daml.org/2003/09/factbook/factbook-ont#>
SELECT DISTINCT ?oil ?name
WHERE
{
        ?country j.1:localShortCountryName ?name .
        {
                ?country j.1:grossDomesticProductPerCapita ?gdp .
                FILTER (?gdp < 1500) .
        }
        UNION
        {
                ?country j.1:grossDomesticProductPerCapita ?gdp .
```

Fig. 19 User interface of SPARQLinG.

The background of the SPARQLinG tool is a large canvas, which can contain a grid – if desired – making it easier to draw and align RDF-GL elements. Elements can be drawn by selecting the appropriate drawing tool and by clicking and dragging on the canvas. SPARQLinG implements a sketch mode, so that users can see a sketch-like representation of an element while holding the mouse, before actually drawing the element (when the mouse is released). Figure 20 shows how a box is drawn in SPARQLinG.

Other features related to drawing RDF-GL queries are moving, resizing, and deleting elements. Whereas boxes can be drawn anywhere on the canvas and their dimensions can be manipulated, arrows can only be drawn from one (valid) element to another – forcing the user to actually touch both elements while drawing the arrow – and their dimensions cannot be changed, since the tool automatically optimizes the location of arrows between two elements. All properties of boxes and arrows can be edited intuitively by means of a property window, which appears when the user clicks on an element.

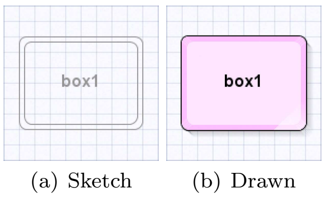
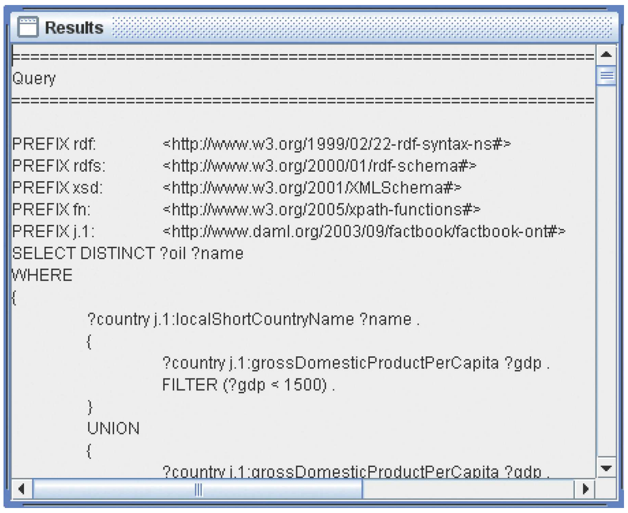


Fig. 20 Drawing a box.

SPARQLinG's features with respect to file input and output are rather basic. Entire RDF-GL drawings can be saved and loaded using dialogs, just like regular graphical applications support saving and loading. Also, drag and drop is supported for loading RDF-GL files. Furthermore, the user can specify the RDF file which is to be queried. After the user specifies the RDF file, the tool automatically parses the file, so that the ontology can be used for drawing RDF-GL queries and queries can be executed immediately. For RDF-GL queries to be executed, the tool is also able to convert RDF-GL to SPARQL. Query results are displayed in a result window, along with the RDF-GL query represented in SPARQL, as shown in Fig. 21.



```
==========================================================
Query
==========================================================

PREFIX rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:       <http://www.w3.org/2001/XMLSchema#>
PREFIX fn:        <http://www.w3.org/2005/xpath-functions#>
PREFIX j.1:       <http://www.daml.org/2003/09/factbook/factbook-ont#>
SELECT DISTINCT ?oil ?name
WHERE
{
        ?country j.1:localShortCountryName ?name .
        {
                ?country j.1:grossDomesticProductPerCapita ?gdp .
                FILTER (?gdp < 1500) .
        }
        UNION
        {
                ?country j.1:grossDomesticProductPerCapita ?gdp .
```

Fig. 21 Results of an executed RDF-GL query.

Despite the fact that prefixes currently are not fully supported in RDF-GL, SPARQLinG automatically assigns a prefix to the ontology used and to default RDF, RDFS, and XSD elements, to make it easier for users to browse through the available IRIs and to ensure readability of the diagrams. In case full paths (IRIs) are used, labels would get hard to read. Future versions of RDF-GL are likely to support prefixes and thus this functionality eventually will become obsolete.

4.3 Experiments with RDF-GL and SPARQLinG

A usability experiment held under a small group of students with fair knowledge on SPARQL querying shows that the combination of RDF-GL and an RDF-GL editor such as SPARQLinG enables one to create and execute complex queries in a convenient and intuitive way. The participants are chosen randomly from a group of students who are indicative of a cross-section of potential end users.

The participants are given a complex query related to the CIA World Factbook ontology (as described earlier), which they need to translate into a SPARQL query and an RDF-GL query. Performance is measured with how much time each user needs to complete each of the two queries. Also, accuracy is measured by means of the number of mistakes each member of the test group makes.

The students need to query the CIA World Factbook for countries which have an import or export to neighbors worth more than $10,000,000,000 a year. The query needs to return the names of both countries and their neighboring trading partners, as well as the percentages of imports and exports and optionally, the inflation rate of the neighboring partners. Only the first 20 results are desired and should be ordered by country name (ascending). In SPARQL, a query which returns the requested results is given in Fig. 22, whereas its RDF-GL query equivalent is presented in Fig. 23.

Results show that about 60% of the students state that creating a complex query using RDF-GL takes (slightly) less time than manually inserting a SPARQL query (for SPARQL experts). Converting the (natural language) search assignment to a valid query takes about as much time with both query languages, but actually drawing this query in RDF-GL sometimes is more time consuming than manually inserting a SPARQL query. The SPARQLinG or RDF-GL user especially benefits from the expressive power of RDF-GL when reusing variables, changing or adding relations between variables, and changing query characteristics (e.g., query type, variables to select). The more complex a query is, the more a user can benefit from RDF-GL over SPARQL.

Although manually inserting a SPARQL query might be faster than drawing an RDF-GL query in some cases, about 80% of the participants indicate that querying becomes easier to do with RDF-GL, because a clear overview

```
PREFIX j.1: <http://www.daml.org/2003/09/factbook/factbook-ont#>
SELECT DISTINCT ?nameC ?nameN ?percentExp ?percentImp ?inflation
WHERE
{
    ?country j.1:conventionalShortCountryName ?nameC .
    ?country j.1:border ?border .
    ?border j.1:country ?neighbor .
    ?neighbor j.1:conventionalShortCountryName ?nameN .
    ?country j.1:exportPartner ?partnerExp .
    ?partnerExp j.1:percent ?percentExp .
    ?partnerExp j.1:country ?neighbor .
    ?country j.1:importPartner ?partnerImp .
    ?partnerImp j.1:percent ?percentImp .
    ?partnerImp j.1:country ?neighbor .
    {
        ?country j.1:imports ?imports .
        FILTER (?imports > 10000000000) .
    }
    UNION
    {
        ?country j.1:exports ?exports .
        FILTER (?exports > 10000000000) .
    }
    OPTIONAL
    {
        ?neighbor j.1:inflationRate ?inflation .
    }
}
ORDER BY ASC(?nameC) LIMIT 20
```

Fig. 22 Complex SPARQL query.

of the complex construction of the query can easily be maintained, since an RDF-GL query usually gives more insight in relations between variables and the entire construction of the query. Problems with respect to easily understanding the expected results of a query and the way a query is constructed will arise when complex queries in SPARQL syntax become larger, whereas RDF-GL's symbols support the understanding and the construction of the query visually, which is more natural for the average end user.

The same 80% of participants that state that querying becomes easier to do using RDF-GL, indicate that the SPARQLinG editor simplifies query creation, because it only allows syntactical correct drawing actions, so that (drawing) errors related to RDF-GL elements occur less. Furthermore, the SPARQLinG editor's functionality of offering all available IRIs from the ontology which is being used, is deemed valuable, as well as the ease with which one can edit the properties of the RDF-GL elements.

All participants agree on the fact that SPARQLinG is able to convert the RDF-GL query to a SPARQL query within acceptable time (less than one second). It should be noted that, while converting RDF-GL to SPARQL and querying the RDF file is done very fast, emptying large result buffers could take up quite some time. Furthermore, the students state that the interface of the tool runs smoothly and works intuitively enough to let a user with fair knowledge of SPARQL be able to draw his or her first RDF-GL query in only a few minutes.

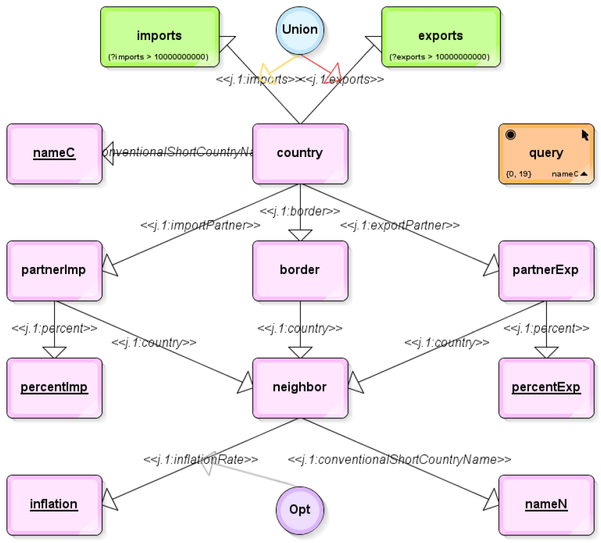


Fig. 23 Complex RDF-GL query.

5 Conclusions and Further Research

The main aim of RDF-GL is to cover as much of SPARQL expressivity as possible while maintaining simplicity and intuitiveness. For best results, a graphical query language such as RDF-GL should be combined with a tool, such as SPARQLinG. This way, complexity of a textual query language (i.e., SPARQL) is hidden by using symbols, text and menus. Not every aspect of a textual query language can be covered by symbols of a graphical query language, and thus some text elements have to be added to the GQL. Drawing (recognizable) query elements is difficult and this is where the user interface comes to play a major part. A user interface should offer the user convenient menus and windows to edit properties of symbols in a query. The combination of RDF-GL and SPARQLinG, makes one able to create and execute complex queries in a convenient and intuitive way.

RDF-GL is the first graphical query language based on SPARQL, designed for RDF. The focus of the language is on SPARQL SELECT queries. Although RDF-GL can handle almost every SELECT query, it currently offers no support for FROM, FROM NAMED and GRAPH elements. However, the design of RDF-GL allows for extensions, and this should form the main focus of future research.

For the design of graphical RDF-GL queries, we have introduced the SPARQLinG application, a Java-based framework that comprises all the required components for the design as well as the generation of queries on any RDF data sources. Currently, the editor lacks a converter from SPARQL queries to RDF-GL queries, which is to be investigated in further research. Naturally, any syntactic/semantic extension of RDF-GL should be mirrored in the application, and this constitutes an inevitable attention point of future development.

References

1. Nikos Athanasis, Vassilis Christophides, and Dimitris Kotzinos. Generating On the Fly Queries for the Semantic Web: The ICS-FORTH Graphical RQL Interface (GRQL). In *International Semantic Web Conference (ISWC 2004)*, pages 486–501. Springer, 2004.
2. Tim Berners-Lee. Notation 3 (N3) A readable RDF syntax, 1998.
3. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.
4. Jethro Borsje and Hanno Embregts. Graphical Query Composition and Natural Language Processing in an RDF Visualization Interface. Bachelor Thesis, Erasmus University Rotterdam, 2006. `http://www.jborsje.nl/publications/bachelor-thesis.pdf`.
5. Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema - W3C Recommendation 10 February 2004, 2004.
6. Tiziana Catarci, Paolo Dongilli, Tania Di Mascio, Enrico Franconi, Giuseppe Santucci, and Sergio Tessaris. An Ontology Based Visual Tool for Query Formulation Support. In *Sixteenth European Conference on Artifical Intelligence (ECAI 2004)*, volume 110 of *Frontiers in Artificial Intelligence and Applications*, pages 308–312, Amsterdam, The Netherlands, 2004. IOS Press.
7. Central Intelligence Agency. The CIA World Factbook, 2008. See `https://www.cia.gov/library/publications/the-world-factbook/index.html`, last visited Oct. 2008.
8. Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A Graphical Language for Querying and Reshaping XML Documents. *Computer Networks*, 31(11–16):1171–1187, 1999.
9. Amineh Fadhil and Volker Haarslev. GLOO: A Graphical Query Language for OWL Ontologies. In *OWL: Experience and Directions (OWLED 2006)*. CEUR-WS, 2006.
10. Volker Haarslev, Ralf Möller, and Michael Wessel. Querying the Semantic Web with Racer + nRQL. In *Third International Workshop on Applications of Description Logics (ADI 2004)*. CEUR-WS, 2004.
11. Andreas Harth, Sebastian Ryszard Kruk, and Stefan Decker. Graphical Representation of RDF Queries. In *Fifteenth International Conference on World Wide Web (WWW 2006)*, pages 859–860, New York, NY, USA, 2006. ACM Press.
12. Human-Computer Interaction Lab, University of Maryland. Piccolo, 2007. See `http://www.cs.umd.edu/hcil/jazz/index.shtml`, last visited Oct. 2008.
13. Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Forth Vassilika Vouton, and Michel Scholl. RQL: A Declarative Query Language for RDF. In *Eleventh International World Wide Web Conference (WWW 2002)*, pages 592–603, New York, NY, USA, 2002. ACM Press.
14. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF - W3C Recommendation 15 January 2008, 2008.
15. Roger S. Scowen. Extended BNF – A generic base standard. ISO 14977.
16. SourceForge. Jena, 2008. See `http://jena.sourceforge.net/`, last visited Oct. 2008.
17. SourceForge. JGraph, 2008. See `http://www.jgraph.com/`, last visited Oct. 2008.
18. SourceForge. Prefuse, 2008. See `http://prefuse.org/`, last visited Oct. 2008.
19. Richard Vdovjak, Peter Barna, and Geert-Jan Houben. EROS: Explorer for RDFS-Based Ontologies. In *Eigth International Conference on Intelligent User Interfaces (IUI 2003)*, pages 330–330, New York, NY, USA, 2003. ACM Press.