Web, Graphs and Semantics

Olivier Corby

To cite this version:

Olivier Corby. Web, Graphs and Semantics. International Conference on Conceptual Structures, Jul 2008, Toulouse, France. 10.1007/978-3-540-70596-3_3 . hal-01531149

Web, Graphs and Semantics

Olivier Corby

INRIA Edelweiss Team
2004 route des lucioles - BP 93
FR-06902 Sophia Antipolis cedex
olivier.corby@sophia.inria.fr

Abstract. In this paper we show how Conceptual Graphs (CG) are a powerful metaphor for identifying and understanding the W3C Resource Description Framework. We also presents CG as a target language and graph homomorphism as an abstract machine to interpret/implement RDF/S, SPARQL and Rules. We show that CG components can be used to implement such notions as named graphs and properties as resources. In brief, we think that CG are an excellent framework to progress in the Semantic Web because the W3C now considers that RDF graphs are – along with XML trees – one of the two standard formats for the Web[1].

1 Introduction

Conceptual Graphs were introduced by John F. Sowa in 1976 when he was at IBM [32] and were popularized in his foundational book of 1984 [33].

The Semantic Web was introduced in 1998 by W3C along with the Resource Description Framework (RDF) that enables the description of graphs. Recently, the SPARQL Query Language for RDF was published as a *Recommendation* by W3C. Further, Tim Berners-Lee informally reformulated his vision of the Semantic Web as a "Web of Data" and also as a "Giant Global Graph"[2] (GGG). This thinking lead to a dramatic change in the architecture of WWW: *RDF graphs* and XML trees were both considered as data structures for information sharing on the Web according by the W3C[3].

In this paper our presentation follows as an exercise of storytelling about the work done in the Edelweiss/Acacia team from INRIA with CG for RDF Semantic Web. We show how Conceptual Graphs (CG) were a powerful metaphor for identifying, understanding and implementing the W3C Resource Description Framework. We also present CG as a target language and graph homomorphism and as an abstract machine to interpret/implement RDF/S, SPARQL and Rules. In particular, we would like to show that CG components can be used to implement such notions as named graphs and properties as resources.

[1] Published in the proceedings of the International Conference on Conceptual Structures, ICCS, Toulouse, July 2008. Springer Verlag.
[2] http://dig.csail.mit.edu/breadcrumbs/node/215
[3] http://www.w3.org/Consortium/technology

In short, we believe that CG are a good framework to progress in the Semantic Web because the W3C now considers that RDF graphs are – with XML trees – one of the two standard formats of the Web.

2 History

The Acacia team previously worked on Information Retrieval through Knowledge Models with CG (PhD Thesis of Philippe Martin [26]) and KADS. We evolved from Knowledge Based Systems to Knowledge Engineering for Corporate Memory Management. We were interested in mixing Knowledge Enginering (KE) and Structured Documents and then KE on the Web.

In 1998 we were interested in and studied XML and XSLT for Structured Web documents. In 1999, RDF was published and, thanks to our CG background, we understood that it could be used to implements graphs (CG/RDF) for documents (XML) and we worked on a first mock-up based on Notio [31, 23] in Java. This first mock-up, called Corese for COnceptual REsource Search Engine [10], implemented the first translator from RDF/S to the CG model. In 2000 we had the pleasure to collaborate with Peter Eklund and Philippe Martin on RDF and CG [27].

Then we focussed on Corporate Semantic Web, a mix between Corporate Memory Management and Semantic Web Technologies. We were involved in a European project called Comma for Corporate Memory Management through Agents [18] within which we started to leverage the mock-up into a research prototype.

Ten years later, we have more than 20 running applications using CG/RDF and 5 generic systems based on the technology. We can now acknowledge that CGs were a good metaphor and enabled us to understand and foresee the Semantic Web project and enable us to participate. It was our chance to be members of the CG community and members of INRIA, one of the the founding members of the W3C.

3 CG for RDF

We have proposed a mapping between RDF and CG and extensions to the simple conceptual graph model in order to implement RDF and SPARQL features such as property variables, named graphs, filters and optional parts.

3.1 RDF Schema

We have designed a mapping between RDF and CG, and RDF Schema and CG support. RDF triples are mapped to relations and resources are mapped to concepts. RDFS classes are mapped to concept types, RDF properties are mapped to relation types, domain and range are mapped to relation signature. SubClassOf and subPropertyOf are mapped to concept and property type subsumption respectively. We have designed the type inference algorithm that enables us to create well typed concepts according to their `rdf:type` and to the signatures of their relations. We have implemented some properties of relations such as symmetry, inverse and transitivity.

An interesting feature of RDF Schema is that it follows RDF syntax, i.e. triples made of a resource, a property and a value. Hence, an RDF Schema statement can be understood as a relation in a graph. For example the RDFS triples below:

```
Human subClassOf Primate
Human label 'human'@en
```

can be translated into the graph relations:

```
[Class:Human]
 -(subClassOf)-[Class:Primate]
 -(label)-[Literal:'human'@en]
```

RDF Schema statements loaded in the graph can be seen as *annotations*. They are related to the instances via the `rdf:type` relation.

```
[Human:Jules]-(rdf:type)-[Class:Human]-(subClassOf)-[Class:Primate]
```

Once present in the graph, the RDFS statements represent (reify) the real types that are present in the support. There is no semantics attached to these relations, the semantics comes from the CG support as usual. They are used as proxies for querying purpose. Two occurrences of the same identifier may represent two different entities according to the context, e.g. `Human` identifies a class and an instance.

For example, the query below retrieves instances of classes whose English label contains the string 'human' and hence finds Jules:

```
?x r ?y . ?y rdf:type ?class .
?class rdfs:label ?l
filter(regex(str(?l), 'human' ) && lang(?l) = 'en' )
```

This feature happens to be extremely useful in real applications where we can query the graph and its schema within the same formalism. Once again, the operational semantics w.r.t. graph projection is carried out by the support.

3.2 Type Intersection

One main difference between RDF and CG is that in RDF a resource may have several types whereas in CG a given concept has but one type. We solved this problem by assigning as a concept type the intersection of the types. Hence, we had to design an algorithm that computes, on the fly, the intersection of two types.

```
x rdf:type T1         p1 domain T1
x rdf:type T2         p2 domain T2
=>                    x p1 y
[T1 AND T2 : x]       x p2 z
                     =>
                     [T1 AND T2 : x]
```

The algorithm maintains the consistency in the type hierarchy. Which means that subtypes of types for which we compute an intersection must then be subtypes of this intersection. In the example below, Aircraft must be a subclass of the intersection of Mobile and Object:

```
Flying   subClassOf Mobile      Aircraft subClassOf Flying
Artefact subClassOf Object      Aircraft subClassOf Artefact

Mobile_AND_Object subClassOf Mobile
Mobile_AND_Object subClassOf Object
=>
Aircraft subClassOf Mobile_AND_Object
```

The intersection algorithm also takes into account disjoint types that cannot generate intersections in their descendants.

3.3 Datatype Values

In order to implement RDF we had to design a datatype extension. Some nodes in the graph carry datatype values such as (integer, 45) or (string, 'Garfield'). Datatype values are implemented as Java objects whose classes implement operators, such as *equal, greater than*, etc., through method overloading. Markers of literal nodes contain such Java objects.

Two input strings may lead to the same datatype value:

`'01'xsd:integer` and `'1'xsd:integer` represent the same value. Hence they must be mapped to the same marker containing the same value.

Operators are implemented through method overloading that realizes type checking, i.e. numbers can compare with numbers, strings with strings, etc. We decided for efficiency reasons to rely on Java polymorphism to tackle type checking.

3.4 Property Concept

In a standard query graph, there may be generic markers associated to concepts but not with relations. Property variables enable the use a variable in a query in place of a property (relation). For example in the query below we search two concepts, `?x` and `?y`, related by any property, denoted by variable `?p`.

```
?x ?p ?y
```

The advantage of using a variable is that we can retrieve the property in the result by getting the value of variable `?p` just as any other variable (e.g. `?x`). In addition, we can search for concepts that are related by the *same* property by using the same variable.

```
?x ?p ?y . ?y ?p ?z
```

Eventually, we can express constraints on the property by means of the variable. For example, we can look for transitive properties:

```
?x ?p ?y . ?p rdf:type owl:TransitiveProperty
```

Or we can search for properties from a specific ontology:

```
?x ?p ?y
filter(regex(str(?p), 'http://www.inria.fr/edelweiss/schema#'))
```

In order to implement the processing of property variable within standard graph projection, we have proposed to reify the property by an additional concept. This concept is of type `rdf:Property` and its marker is the name of property. Each occurrence of relation in a graph contains the additional concept that represents (reify) the property.

```
x1 r y1 -> r(x1, y1, r)
x2 q y2 -> q(x2, y2, q)
```

Hence we manage hyperarcs, i.e. arcs that relate more that two nodes. It is remarkable that several authors [3, 14, 22] propose the same extension from a theoretical point of view.

In our extension, a query relation may or may not use a property variable. If not, the property concept is invisible and is not processed during graph projection.

3.5 Named Graph

Following the same design pattern, we have implemented a second extension for named graph. A named graph is a graph which is associated a name by means of a URI. This URI is a standard resource that can itself be annotated by means of properties.

In the example below, `g1` is the name of a graph:

```
g1 { cat on mat . cat name 'Garfield' }
g1 author James
```

The name of the graph (the URI) is reified as an additional concept and each relation of a given graph contains this additional concept. With the example above, and with `std` as the URI of the standard graph (the graph with no name):

```
on(cat, mat, on, g1)
name(cat, 'Garfield', name, g1)
author(g1, James, author, std)
```

Note that `name` (resp. `author`) appears once as the name of the relation and once as the concept that reifies the relation, according to the hyperarc point of view explained above. Hence, the same name is used for different entities.

We are then able to process queries with graph patterns by matching the graph URI with the additional argument carried by the hyperarcs:

```
select * where {
 graph ?g { cat on ?place }
}
```

This query is translated into the following hyperarc where `_:b` represents a query blank which means that we don't care about the property concept:

```
on(cat, ?place, _:b, ?g)
```

We obtain as result:

```
?g = g1 ; ?place = mat
```

It is remarkable that this very simple idea, implementing named graphs with an additional argument, solves the problem of representing *and* querying named graphs. This is what we mean by considering CG as a valuable target abstract machine to implement RDF processing. The SPARQL *from* and *from named* clauses are implemented by adding appropriate filters on the graph variables.

```
select *
from <g1>                      on(?cat, ?place, on, ?g)
where { ?cat on ?place }       filter(?g = <g1>)
```

3.6 Inference Rules

We have designed a forward chaining graph rule language with an RDF/SPARQL syntax. This language is inspired by Salvat and Mugnier [30]. The syntax of the rule condition and conclusion patterns is that of SPARQL patterns (i.e. collections of triples). We have included the graph pattern in the syntax, hence it is possible to take named graphs into account.

```
graph ?g { ?x ?p ?y . ?y rdf:type owl:SymmetricProperty }
=>
graph ?g { ?y ?p ?x }
```

3.7 Projection

We have designed and implemented an hypergraph homomorphism algorithm based on relation enumerations following heuristics to optimize the search. The order in which the query relations are considered is compiled according to heuristics such as the relation's cardinality (number of occurrences), connexity, presence of filters, etc. In addition to compiling the order of query relations, the algorithm is able to backjump in case of a failure due to the absence of a target relation or due to the failure of a constraint. By backjump we mean that it is able to backtrack – not systematically to the preceding query relation – but to a preceding query relation that may solve the failure. The index of where to backjump is determined statically and compiled.

In addition to property variables and graph patterns for named graphs, the algorithm is able to process optional query parts. If an optional part fails, the query does not fail. If it succeeds, the answer contains additional information.

Example: retrieve resources which have a name (mandatory) and which may have an age (optional).

```
[?x]-(name)-[?name]
optional { [?x]-(age)-[?age] }
```

An optional part may contain several relations, in which case it succeeds if *all* relations succeed. It may contain filters in which case it succeeds if the filters evaluate to true. It may contain nested optional parts which are processed only if the current optional part succeeds. Hence, the processing of queries with optional parts imply the introduction of scopes surrounding the optional parts.

Eventually, the algorithm has been adapted to interpret SPARQL queries with *select, distinct, order by* and *limit* operations. The *distinct* operation is an interesting constraint that ensures that two answers do not contain the same variable bindings, e.g. `select distinct ?x ?y` ensures that the bindings of *?x, ?y* differ in all answers. Hence, we need to manage a list of current answers to the homomorphism and check that the current answer that is computed is distinct from all previous answers. An optimization computes the distinct set as soon as all variables are bound in the partial result. If it is not the case, the graph homomorphism backtracks and searches for other bindings. In practice, the algorithm *backjumps* to a new binding.

In addition, we have added a *group by* operation that enables us to group results that share same variable binding for some variables and *count()* that enables to count the number of values of a variable after grouping. We have also added the possibility of returning the result of an expression in the result (in the select clause). For example, the query below retrieves persons that are the authors of documents, groups the results by person, counts the documents of each author and returns the counter in the result.

```
select ?person count(?doc) as ?count
where { ?person :author ?doc }
group by ?person
```

These operations fits smoothly within graph homomorphism but the SPARQL union operation does not fit well into this paradigm. It needs to be implemented as an operator of an interpreter that would implement AND, UNION and OPTIONAL operations applied to elementary graph homomorphisms.

3.8 Constraints

Another originality of our homomorphism algorithm is that it is able to take additional constraints on node values into account. Example of constraints are: `?x != ?y`, `?date <= '2008-01-01'` and `fun:foo(?x, ?y)` where `?x, ?y, ?date` represent the value of the target nodes associated by homomorphism to the query nodes denoted by the variables.

A query graph with constraint matches a target subgraph found by homomorphism if the constraint evaluates to true when applied to the appropriate nodes of the target graph. Constraints are prefixed by the `filter` keyword.

Examples:

```
[?x]-(r)-[?z]-(p)-[?y]   filter(?x != ?y)

[?x]-(birth)-[?date]     filter(?date <= '2008-01-01')
```

We have designed a constraint language that has been extended to process SPARQL filters. The language enables us to define simple operations such as comparisons between node values: `?x != ?y`, boolean expressions such as: `?x != ?y && ?z <= '2004-01-01'` and function calls such as: `xsd:datatype(?x)`.

The atomic entities of the language are constants and variables. Constants are values carried by the nodes of the target graph. They may be URIs of resources or literal values such as strings, integers, booleans and dates. Variables represent the values of target nodes found by graph homomorphism. Values of target nodes are datatype objects, similar to the constants, that implement polymorphic operators according to type checking rules (integers do not compare with strings, floats compare with doubles, etc.).

Constraint expressions (`EXP`) are built on top of the atomic entities (`CST, VAR`) with function calls (`FUN`) and terms (`TERM`). Terms are recursively build with expressions related by operators. An abstract syntax of the constraint language is given below:

```
EXP  ::= CST | VAR | FUN | TERM
FUN  ::= NAME ( EXP* )
TERM ::= EXP and EXP | EXP or EXP | not EXP |
         ( EXP )     | EXP OPER EXP
OPER ::= <  <=  =  !=  >=  >  +  -  *  /
```

The projection algorithm cooperates with a constraint evaluator that is able to evaluate partial constraints according to a current partial binding. As soon as the variables of a constraint are bound by target nodes, the constraint is evaluated. If the expression evaluates to true, the projection continues (the current partial projection is successful). If it fails, the projection algorithm backtracks in order to find another binding for the variables. In fact, the algorithm *backjumps* in order to effectively change the binding.

The evaluator is a recursive function that has two arguments: an expression of the constraint language and an environment that contains variable bindings. Variable bindings are computed by the projection and are the values of the target nodes corresponding to the query node variables, e.g.
`?x = 12 ; ?y = '2007-01-01' ; ?z = URI`. The evaluator returns values of the same domains as the constants. The final result of a constraint evaluation must evaluate to true.

A scheme of the constraint evaluator is given below where `exp` is the constraint expression and `env` is the variable binding environment.

```
eval(exp, env){
switch(exp){
 case constant : return exp;
 case variable : return env.get(exp);
 case funcall  : values = for all arg(exp) : eval(arg, env);
                 return apply(fun(exp), values);
 case not      : return ! eval(arg(exp), env);
 default : return apply(operator(exp), eval(arg1(exp), env),
                                       eval(arg2(exp), env));}}
```

Complex constraint expressions are decomposed into smaller ones which are associated to subpart of the query where their variables are bound and they are evaluated as soon as possible in order to cut the search tree.

It must be noted that – as in SPARQL – it is possible to test a *negation as failure* query using an optional pattern and a `! bound()` constraint. As an example, the query below searches persons that are *not* author of a document. The query search for an optional `author` relation. If it is not found, the query succeeds; if it is found, the constraint fails because the `?doc` variable is bound and hence the query fails.

```
select * where {
 ?x rdf:type :Person
 optional { ?x :author ?doc }
 filter(! bound(?doc))
}
```

3.9 Type Relaxation

Our projection algorithm is able to perform approximate search wrt types. It is possible to relax type checking according to subsumption. For example, when searching for a person author of an article, we may return a research team author of a report. We relax the type Person by Team and the type Article by Report. We compute a semantic distance between concept types which decreases with depth like in [35] and try to minimize the sum of the distances.

This idea happens to be quite interesting and we have generalized this relaxation process. It is now possible to design and program a new distance algorithm and specify such a user defined algorithm in a query. Hence, the user can try different relaxation algorithms according to the domain and/or the query. The syntax is the following where the `more` keyword authorizes relaxation, the prefix specifies where to fin the Java package of the user defined distance algorithm and the `relax by` statement requires the user defined distance.

```
prefix dd: <fun://fr.inria.edelweiss.Distance>
select more * where { PATTERN }
relax by dd:distance
```

3.10 Graph Path

We implemented an extension to SPARQL to process path queries, inspired by [25, 1]. The path algorithm avoids cycles. Using a path variable in place of the property is done by introducing a $ prefixed variable, which means find a path of one or more relations that links `a` and `b`:

```
a $path b
```

It is possible to test the length of the target path:

```
pathLength($path) >= 2 && pathLength($path) <= 8
```

It is possible to associate a regular expression that must be matched by the types of the relations of the target path. In the following case, we want the properties to be either `p1` or `p2`. By default, we also accept subproperties.

```
match($path, star(p1 || p2))
```

We have designed and implemented the following original extension in order to match the target relations that have been found in the path. The target relations of the path are grouped in a transient *named graph* whose name is given by the path variable. Hence this named graph is accessible by means of a graph pattern on the path variable. It has for effect to enumerate the target path relations as shown below, where `$path` is the path variable:

```
graph $path { ?x ?p ?y }
```

The purpose of this pattern, in addition to enumerate the path relations in the result, is to enable us to specify additional constraints such as in the examples shown below. For instance, to go through a specific resource within the path:

```
graph $path { ?x ?p ?y  filter(?x = a || ?y = a) }
```

Or not to go through a specific resource:

```
graph $path {
 optional { ?x ?p ?y  filter(?x = a || ?y = a) }
 filter (! bound(?p))
}
```

Or to find a specific pattern within the path:

```
graph $path { ?x p a . a q ?z }
```

This path algorithm has been applied to the Insee RDF base that describes French territory[4] with 500,000 relations and a version which computes shortest path was able to find a shortest path between Nice and Grenoble in 0.3 sec.

Another extension of the path algorithm for navigating through recursively nested contexts is explained below.

4 Context

Recently, we have been working on contexts using named graphs. A named graph is a graph which has a name given by a URI and which is accessible by means of a graph pattern in a query. In addition, this URI is itself a resource that can be part of graphs.

Two named graphs, `g1` and `g2`, are shown below:

```
g1 { a p b . b q c}       g2 { a r d }
```

A query with a graph pattern to retrieve relations in named graphs:

```
select *
from named <g1>
from named <g2>
where {
 graph ?g { ?x ?p ?y }
}
```

A special case of named graphs enables us to describe nested graphs such as nested Conceptual Graphs.

```
:Alice c:tell :story
:story { :Cat :on :Mat }
```

Named graphs and graph patterns are simple but powerful notions that enable us to model contextual metadata where a context is a named graph and is denoted by its name.

[4] http://rdf.insee.fr/geo/

4.1 Hierarchy of type of context

It is possible to model a hierarchy of class of context and to type the URI of the named graphs. Hence we can retrieve contextual metadata according to context types and exploit subsumption.

```
Past subClassOf Context
 Prehistory subClassOf Past
   Paleolithic subClassOf Prehistory
   Neolithic subClassOf Prehistory
Present subClassOf Context
Future  subClassOf Context

g1 rdf:type :Paleolithic
g1 { :man :practice :hunting }

g2 rdf:type :Neolithic
g2 { :man :practice :agriculture }
```

A query that retrieves activities in contexts of type Prehistory, i.e. Paleolithic and Neolithic:

```
graph ?g { :man :practice ?activity }
?g rdf:type :Prehistory
```

4.2 Annotation of context

Thanks to its uniform nature, it is possible to annotate context by means of its name which is a URI.

```
g1 rdf:type :Paleolithic     g2 rdf:type :Neolithic
g1 :start -2500000           g2 :start -10000
g1 :location :Europe         g2 :location :MiddleEast
```

Note that it is possible to have several contexts of type Neolithic that start at different dates according to the location. We can then query contextual metadata:

```
graph ?g { ?x :practice ?activity }
?g :start ?date filter(?date <= -10000)
```

4.3 Contextual relations

We can now model semantic relations between contexts such as temporal relations. It is possible to define spatio/temporal relations, linguistic relations such as those used in rhetorical structure theory (RST), logical relations, etc. Note that contextual relations can themselves be contextualized. For example, `g1 sequence g2` is true in context `state1`:

```
g1 { ... }     state1 { g1 sequence g2 }
g2 { ... }     state2 { g3 sequence g4 }
g3 { ... }     state3 { state1 parallel state2 }
g4 { ... }
```

We can then query what happens in a context `?g2` after a given context `?g1`:

```
graph ?g1 { ?x ?p ?y }
graph ?g2 { ?z ?q ?t }
?g1 sequence ?g2
```

It is of course desirable to specify the algebraic properties of the contextual relations, e.g. parallel is symmetric and transitive, sequence is transitive, etc. This can be done using OWL light statements that are interpreted in Corese. More complex algebraic properties of relations can be modeled by rules.

```
parallel rdf:type owl:SymmetricProperty
parallel rdf:type owl:TransitiveProperty
sequence rdf:type owl:TransitiveProperty
```

4.4 Rec graph pattern

In order to enable querying contextual relations, we have designed a generalized version of the path algorithm dedicated to nested contexts.

In the example above, suppose that we want to search/retrieve triples recursively nested within the state3 context, i.e. the triples in state1, state2, g1 and g2. We need to know the exact relations between the nested context to retrieve these triples. It may be impossible to be aware of the whole structure. To solve this, we propose a new query pattern called `rec graph` (recursive graph) as follows:

```
rec graph state3 { ?x ?p ?y }
```

The result of the query will be the triples from the state3 graph and the triples from the recursively nested graphs, e.g. state1 and g1. This is computed by the path algorithm described above. Instead of searching for path from ?x to ?y (first and second arguments), the algorithm searches for path from state3 to ?x, (i.e. from graph name argument to first argument). An example of such a path of length 3 is: `(state3, state1), (state1, g1), (g1, a)` as shown below:

```
(1) state3 { state1 parallel state2 }
(2) state1 { g1 sequence g2 }
(3) g1 { a p b }
```

Another path would be: `(state3, state2), (state2, g4), (g4, b)`. This query pattern also enables to search if several triples are recursively related by an embedding context:

```
rec graph ?g { a p b . c q d }
```

It is also possible to specify a regular expression on the relations that link the nested contexts as shown below:

```
rec graph ?g {
 ?x ?p ?y
 filter(match(?p, star(log:property)))
}
```

4.5 Defining a resource using a named graph

The named graph statement enables to assign a name (a URI) to a graph. We propose to use this statement in a slightly different way in order to assign a definition (a graph) to a URI (it's name). This enables to define composite objects made of atomic objects where none of the sub objects plays a special role. Hence we assign a URI to a composite structure made of several related objects. The URI can then be used in other composite structures.

For example, we define the H_2O molecule as a named graph containing a description of two hydrogens related to one oxygen. The `cos:graph` attribute is a syntactic extension to RDF/XML, (W3C member submission [19]), that enables to define the URI of a named graph. Note that in the example below, there are two different Hydrogens (two blank nodes) related to the same Oxygen (one blank node with 'o' ID).

```
<c:Hydrogen cos:graph='&c;H2O' >
  <c:related><c:Oxygen rdf:nodeID='o'/></c:related>
</c:Hydrogen>
<c:Hydrogen cos:graph='&c;H2O' >
  <c:related><c:Oxygen rdf:nodeID='o'/></c:related>
</c:Hydrogen>
```

This RDF description is equivalent to the named graph:

```
H2O { [H]-(r)->[O]<-(r)-[H] }
```

Then we define the CH_4 molecule as a named graph containing a description of one carbon related to four hydrogens.

```
<c:Carbon cos:graph='&c;CH4' >
  <c:related><c:Hydrogen/></c:related>
  <c:related><c:Hydrogen/></c:related>
  <c:related><c:Hydrogen/></c:related>
  <c:related><c:Hydrogen/></c:related>
</c:Carbon>
```

We can now query the structure of a molecule using a named graph pattern.

```
select ?atom  countItem(?at) as ?count where {
   graph c:H2O { ?at rdf:type ?atom }
}
group by ?atom

atom = H ; count = 2
atom = O ; count = 1
```

We can then define a product *Prod* as a named graph containing two molecules, one instance of H_2O and one instance of CH_4. Note that molecules are now considered as classes that are instantiated. We could also use a property to relate a molecule to its definition as H_2O, e.g. `_:b :definition c:H2O`.

```
<c:H2O cos:graph='&c;Prod' />
<c:CH4 cos:graph='&c;Prod' />
```

Then we define an instance of *Prod* that will hence contain one H_2O and one CH_4.

```
<c:Prod cos:graph='&c;exp' />
```

Queries

We can now write queries to check the structure of the product and of the molecules. The graph pattern query below destructures the named graph in order to retrieve the molecules that compose the product `c:Prod`. The `c:isTypeOf` property is the inverse of `rdf:type`.

```
select  ?part countItem(?p) as ?count where {
  graph c:Prod  { ?part c:isTypeOf ?p }
}
group by ?part
```

The result is:

```
part = H2O ; count = 1
part = CH4 ; count = 1
```

The recursive graph pattern query below (`rec graph`) recursively destructures the named graphs in order to retrieve the molecules and the atoms that compose the product `c:Prod`. Note that the query is the same as the one above except that we have added the keyword `rec`.

```
select ?part countItem(?p) as ?count where {
  rec graph c:Prod  { ?part c:isTypeOf ?p }
}
group by ?part
```

The result is:

```
part = H2O ; count = 1            part = H   ; count = 6
part = CH4 ; count = 1            part = O   ; count = 1
                                  part = C   ; count = 1
```

This example shows the power of representing a composite object through a URI of named graphs as we have the inverse operation that enables us to walk through the internal structure recursively by means of the `rec graph` pattern.

5 Applications

In this section we show that the couple CG/RDF has proved to be a very fruitful idea in term of systems and applications.

5.1 Generic Systems

There are now several generic CG/RDF based systems that have been designed and developed in the Edelweiss team:

Corese[5] is a generic RDF/S, SPARQL & Rules Semantic Factory that is entirely based on CGs and where CGs are the abstract machine which implements RDF graph operations by means of graph homomorphism.

Sewese[6] is a Semantic Web Server Platform based on Tomcat and Java Taglib [17]. Sewese is built on the Corese engine and provides a set of primitives to build interfaces for queries, edition and navigation, and for the management of the transverse functions of a portal (presentation, internationalization, security, etc.). An ontology editor, a generic annotation editor and a basic rule editor are parts of the Sewese platform. The main purpose of Sewese is to integrate recurrent semantic web operations (e.g. perform a SPARQL Query, transform a result binding in a given view) within a classic web technology framework (e.g. JSP pages, servlet calls).

SweetWiki[7] is a wiki built around a semantic web server that uses semantic web technologies to support and ease the life cycle of wikis [4]. It implements folksonomy based navigation into the wiki pages.

Ecco is a Cooperative Ontology Editor dedicated to support end-users with different profiles (domain expert, engineer, ontologist, ...) in a cooperative process of ontology construction and evolution. The Ontology is managed by the Corese Factory.

SemAnnot is Generic platform for annotation extraction from text using NLP parsers and ontologies [24], the ontology and annotation processor is also Corese.

[5] http://www-sop.inria.fr/edelweiss/wiki/wakka.php?wiki=Corese
[6] http://www-sop.inria.fr/edelweiss/wiki/wakka.php?wiki=Sewese
[7] http://www-sop.inria.fr/edelweiss/wiki/wakka.php?wiki=SweetWiki

5.2 Applications

We have been involved in more than 20 applications that use the RDF/CG mapping.

SevenPro[8] is an European project on Semantic Virtual Engineering Environment for Product Design. Corese is used as Semantic Engine for Text mining and Virtual Reality annotation.

e-WOK[9] is a french ANR project that aims at designing a Semantic Web Platform for Geo Sciences. It aims at building a set of communicating portals (called e-WOK Hubs), offering both: (a) web applications accessible to end-users through online interfaces, and (b) web services accessible to applications through programmatic interfaces As applicative objectives, e-WOK aims at enabling the management of the memory of several projects on CO_2 capture and storage, with use of results of technological watch on the domain.

Two projects focus on semantic text mining of scientific literature in biology. SeaLife[10] is an European project on "A Semantic Grid Browser for the Life Sciences Applied to the Study of Infectious Diseases". ImmunoSearch[11] is a French project on searching biomarkers for controlling and maintaining the harmlessness of molecules used in perfumes, aromatics and cosmetics.

The Palette[12] European project is about "Pedagogically sustained Adaptative LEarning Through the exploitation of Tacit and Explicit Knowledge". It aims at designing semantic web services to help communities of practice communicate and share knowledge.

In the past we have also worked on Knowledge Management Platforms[21] (KMP and KM2) and on Corporate Memory Management through Agents (Comma). There are also projects that we are not members of and that make use of Corese. For example, Neurolog[13] is an ANR Funded project on Medical Imaging with Software technologies for integration of process, data and knowledge in medical imaging.

6 Conclusion

Semantic Web and RDF provide a unique opportunity to use CGs in large scale applications. The basic idea is to consider RDF/S as an input format to build CGs and hence test CG algorithms on large scale real applications. In effect, it is now possible to load schema and data from all over the world as more and more RDF Schema and RDF metadata are available online.

We have shown that it is possible to mix several languages – among which are RDF/S, SPARQL and its XML Result format, RDF Rule language and CGs.

[8] http://www.sevenpro.org
[9] http://www-sop.inria.fr/edelweiss/projects/ewok
[10] http://www.biotec.tu-dresden.de/sealife
[11] http://www-sop.inria.fr/edelweiss/wiki/wakka.php?wiki=Projects
[12] http://palette.ercim.org
[13] http://neurolog.polytech.unice.fr/doku.php

In addition, XSLT can be used for interoperation and presentation. Our work demonstrates that CG technology can be integrated into a complex software system. The system itself can then be used to build various applications such as Semantic Wikis, Semantic engine for an ontology based natural language processing platform or a Virtual Reality semantic engine. The point is to rely on standard languages for input/output, to focus on information and knowledge retrieval and not on presentation or editing issues within the semantic engine itself. Presentation is delegated to external processors such as XSLT engines and web servers. CG were highly successful for understanding and implementing RDF and SPARQL. The only hard problem that we encountered was the SPARQL UNION operator. We have also shown that interesting performance can be obtained – we answer in less than half a second to queries to a graph with 500,000 relations (the insee RDF base[14]).

Further, we identify open problems that would be interesting to tackle within a mix CG/RDF viewpoint: library of semantic distances, scaling to graphs with some giga-relations, indexing such giant graphs, processing queries by distributed graph homomorphism.

To finish, it has always been a great surprise that so little work on the Semantic Web makes use of CGs.

Acknowledgement

This work was performed in the Edelweiss (formerly Acacia) team at the INRIA Sophia Antipolis - Méditerranée. We would like to thank all our colleagues, among whom: Rose Dieng-Kuntz, Alain Giboin, Fabien Gandon, Jean-François Baget, Priscille Durville, Khaled Khelif, Hacène Cherfi, Virginie Bottollier, Olivier Savoie, Francis Avnaim as well as Catherine Faron-Zucker and Michel Buffa from University of Nice-Sophia Antipolis and Philippe Martin.

References

1. Faisal Alkhateeb, Jean-Franois Baget, and Jérôme Euzenat. RDF with Regular Expressions. Technical Report RR-6191. `http://hal.inria.fr/inria-00144922/en`.
2. Mekafor Anyanwu, Angela Maduko, and Amit Sheth. SPARQL2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In *Proc. WWW 2007*, Banff, Alberta, Canada, May 2007.
3. Jean-François Baget. RDF Entailment as a Graph Homomorphism. In *Proc. of the 4th International Semantic Web Conference (ISWC'2005)*, number LNCS 3729, pages 82–96, Galway, Ireland, 2005.
4. Michel Buffa, Fabien Gandon, and Guillaume Erétéo. A Wiki on the Semantic Web. In Jorg Rech, Bjorn Decker, and Eric Ras, editors, *Emerging Technologies for Semantic Web Environments: Techniques, Methods and Applications.* Fraunhofer Institute for Experimental Software Engineering (IESE), Germany, July 2007.
5. Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named Graphs, Provenance and Trust. In *Proc. of WWW 2005*, Chiba, Japan, 2005.

[14] http://rdf.insee.fr/geo/

6. Michel Chein and Marie-Laure Mugnier. Conceptual Graphs: Fundamental Notions. *Revue d'Intelligence Artificielle*, 6(4):365–406, April 1992.
7. Olivier Corby, Rose Dieng-Kuntz, and Catherine Faron-Zucker. Querying the Semantic Web with Corese Search Engine. In R. Lopez de Mantaras and L. Saitta, editors, *Proc. of the 16th European Conference on Artificial Intelligence (ECAI'2004), Prestigious Applications of Intelligent Systems*, pages 705–709, Valencia, Spain, August 22-27 2004.
8. Olivier Corby, Rose Dieng-Kuntz, Catherine Faron-Zucker, and Fabien Gandon. Ontology-based Approximate Query Processing for Searching the Semantic Web with Corese. Research Report 5621, INRIA, 2005.
9. Olivier Corby, Rose Dieng-Kuntz, Catherine Faron-Zucker, and Fabien Gandon. Searching the Semantic Web: Approximate Query Processing based on Ontologies. *IEEE Intelligent Systems & their Applications*, 21(1):20–27, January-February 2006.
10. Olivier Corby, Rose Dieng-Kuntz, and Cédric Hebert. A Conceptual Graph Model for W3C Resource Description Framework. In Bernhard Ganter and Guy W. Mineau, editors, *Conceptual Structures: Theory, Tools and Applications, Proc. of the 8th ICCS*, pages 468–482, Darmstadt, Germany, August 13 -17 2000. Springer-Verlag, LNAI n. 1867.
11. Olivier Corby and Catherine Faron-Zucker. Corese: A corporate semantic web engine. In *Proceedings of the International Workshop on Real World RDF and Semantic Web Applications, 11th International World Wide Web Conference*, Hawai, USA, May 7 2002.
12. Olivier Corby and Catherine Faron-Zucker. Implementation of SPARQL Query Language based on Graph Homomorphism. In *Proc. of the 15th ICCS*, pages 472–475, Sheffield, UK, July 2007.
13. Olivier Corby and Catherine Faron-Zucker. RDF/SPARQL Design Pattern for Contextual Metadata. In *Proc. of IEEE/WIC/ACM Int. Conf. on Web Intelligence*, Silicon Valley, USA, November 2007.
14. Frithjof Dau. RDF as Graph-based, Diagrammatic Logic. In *Foundations of Intelligent Systems, Proc 16th International Symposium on Methodologies for Intelligent Systems, ISMIS*, Bari, Italy, 2006. Springer LNAI.
15. Todd Detwiler. GLEEN: Regular Paths for ARQ SparQL. Technical report, University of Washington. `http://sig.biostr.washington.edu/projects/ontviews/gleen`.
16. Rose Dieng-Kuntz and Olivier Corby. Conceptual Graphs for Semantic Web Applications. In *Conceptual Structures: Common Semantics for Sharing Knowledge, Proc. of the 13th ICCS*, pages 19–50, Kassel, Germany, July 17-23 2005. Springer-Verlag, LNAI 3596.
17. Priscille Durville and Fabien Gandon. Sewese : Semantic Web Server. In *WWW'2007 Developers track*, Banff, Canada, 2007.
18. Fabien Gandon. *Distributed Artificial Intelligence And Knowledge Management: Ontologies and Multi-Agents Systems for a Corporate Semantic Web*. PhD thesis, University of Nice-Sophia Antipolis, november 2002.
19. Fabien Gandon, Virginie Bottollier, Olivier Corby, and Priscille Durville. RDF/XML Source Declaration, September 2007. W3C Member Submission, `http://www.w3.org/Submission/rdfsource`.
20. Fabien Gandon, Virginie Bottollier, Olivier Corby, and Priscille Durville. RDF/XML Source Declaration. In *Proc. of IADIS International Conference WWW/Internet*, Vila Real, Portugal, October 2007.

21. Alain Giboin, Fabien Gandon, Nicolas Gronnier, Cécile Guigard, and Olivier Corby. Comment ne pas perdre de vue les usage(r)s dans la construction d'une application à base d'ontologies ? retour d'expérience sur le projet KmP. In M.C. Jaulent, editor, *Actes des 16e Journées francophones d'Ingénierie des connaissances (IC'2005)*, pages 133–144, Grenoble. France, 2005. PUG.
22. Jonathan Hayes and Claudio Gutierrez. Bipartite Graphs as Intermediate Model for RDF. In *Proc. International Semantic Web Conference, ISWC*, 2004.
23. Cédric Hebert. Modèle de traitement de RDF basé sur les graphes conceptuels. Master thesis, I3S, University of Nice-Sophia Antipolis, 1999.
24. Khaled Khelif, Rose Dieng-Kuntz, and Pascal Barbry. An Ontology-based Approach to Support Text Mining and Information Retrieval in the Biological Domain. *Journal of Universal Computer Science (JUCS), Special Issue on Ontologies and their Applications*, 2007.
25. Krys J. Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *Proc. European Semantic Web Conference, ESWC'2007*, Innsbruck, Austria, 2007.
26. Philippe Martin. *Exploitation de graphes conceptuels et de documents structurés et hypertextes pour l'acquisition de connaissances et la recherche d'informations.* PhD thesis, University of Nice-Sophia Antipolis, october 1996.
27. Philippe Martin and Peter Eklund. Conventions for Knowledge Representation via RDF. In *Proc. of WebNet 2000*, San Antonio, Texas, USA, November 2000.
28. Marie-Laure Mugnier and Michel Chein. Représenter des connaissances et raisonner avec des graphes. *Revue d'IA*, 10(1):7–56, 1996.
29. Sebastian Rudolph, Markus Krotzsch, and Pascal Hitzler. Quo Vadis, CS? On the (non)-Impact of Conceptual Structures on the Semantic Web. In *Proc. of the 15th ICCS*, Sheffield, UK, 2007. Springer-Verlag.
30. Eric Salvat and Marie-Laure Mugnier. Sound and Complete Forward and Backward Chainings of Graph Rules. In *Proc. of the 4th ICCS*, pages 248–262, Sydney, Australia, 1996. Springer-Verlag, LNAI, 1115.
31. Finnegan Southey and James G. Linders. Notio - a java API for developing CG tools. In *Proc. ICCS*, pages 262–271, 1999.
32. John Sowa. Conceptual Graphs for a Database Interface. *IBM Journal of Research and Development*, (4):336–357, 1976.
33. John Sowa. *Conceptual Structures - Information Processing in Mind and Machine.* Addison Wesley, 1984.
34. H. Stoermer, I. Palmisano, D. Redavid, L. Iannone, P. Bouquet, and G. Semeraro. RDF and Contexts: Use of SPARQL and Named Graphs to Achieve Contextualization. In *Proc. 1st Jena User Conference*, Bristol, UK, 2006.
35. J. Zhong, H. Zhu, J. Li, and Y. Yu. Conceptual graph matching for semantic search. In *Proc. of 10th ICCS*, pages 92–106, Borovets, Bulgaria, 2002. LNCS 2393, Springer Verlag.