HORUS: a Configurable Reasoner for Dynamic Ontology Management

Giovanni Lorenzo Napoleoni, Maria Teresa Pazienza, Andrea Turbati
Department of Enterprise Engineering
University of Rome Tor Vergata
Rome, Italy
e-mail: giovannilorenzo.napoleoni@gmail.com,
{pazienza, turbati}@info.uniroma2.it

***Abstract*—This paper introduces HORUS (Human-readable Ontology Reasoner Unit System), a configurable reasoner which provides the user the motivations for every inferred knowledge in the context of a reasoning process. We describe the reasoner, how to write an inference rule and check which explicit knowledge was used to infer a new one. Real cases examples will be provided to show the capabilities of our reasoner and the associated language developed to express inference rules. We show how HORUS allows the user to understand the logical process over which each new RDF triple has been generated.**

Keywords—Ontology Management; Reasoner; Formal Language

I. INTRODUCTION AND MOTIVATION

The Semantic Web is becoming more and more popular and easy to work with. Ontologies are used as a common base to all the applications which rely on such a framework. Main features of an ontology are:

- The use of a specified standards, such as Resource Description Framework (RDF) [1];
- The possibility to infer knowledge from existing one.

The process of inferring new knowledge, from the existing one, is delegated to reasoners. They take in input a vocabulary, the data stored in the ontology and a list of rules and produce new knowledge, hopefully in the same standard in which the ontology is written. A list of existing reasoners can be found at [2]. They differentiate for:

- The rules they are able to use in the inference process;
- Under which license they are distributed, inside which tool they can be used;
- The language in which they are written (e.g., Java, C++, etc.);
- The possibility to accept new rules without the need to change most of their source code;
- Performances in the inference process.

Once a reasoner has been chosen, it is possible to use it:

- Standalone as a tool to infer new knowledge that is saved in a particular serialization (with or without the analyzed knowledge base);
- As a component, inside a framework to immediately observe the inferred knowledge.

Generally, the task of visualizing the results of any tool embedded inside a framework is finalized to both validate its output and produce some performance metrics (such as precision or recall). The validation process for a reasoner is very different: in fact as a list of inference rules is used, a reasoner is characterized on which inference is able to run, its scalability regarding the size of the ontology it analyzes and the time it needs to process it.

By analyzing different reasoners, we discover that they can be really optimized regarding the execution time while both customization and visualization processes are generally lacking, even if they are important and useful, as discussed in [3] (see as an example the use of the framework Protégé (version 3.4.8) [4] and the bundled reasoner Pellet (version 1.5.2) [5]). In this case, a user is not able to know immediately which rules the reasoner will apply. The sole possibility is to consult its home page, [6] for Pellet 2.0. New rules can be added using the language SWRL [7]. Protégé 3.4.8 provides inferred knowledge generated from the selected reasoner, specifying that it has been inferred, without showing which underling knowledge was used in the inference process and why such new knowledge has been produced. There are several contexts in which users could be interested to follow the reasoning process, as for:

- Learning how it behaves;
- Comparing results in different application domains;
- Comparing results with his own expectations related to previous/personal conceptualizations.

Protégé 4.3 has a new system to manage reasoners (Protégé 3.x and Protégé 4.x are used depending on the Ontology characteristics and the existing plug-in). It has two bundled reasoners, FaCT++ [8] and HermiT [9]; but other reasoners can be downloaded and installed. It is also able to provide an explanation to why an inferred knowledge has been (temporarily) added to the selected ontology, but this explanation consists of just the list of explicit triples used by the current reasoner, without showing the other inferred knowledge produced and successively used along the reasoning process, so for complex reasoning it can be difficult to follow the entire process.

For all these reasons, we decided to develop a new reasoner characterized by the following features:

- Being open source;
- Implemented as a Java library;

 ISBN: 978-1-61208-340-7

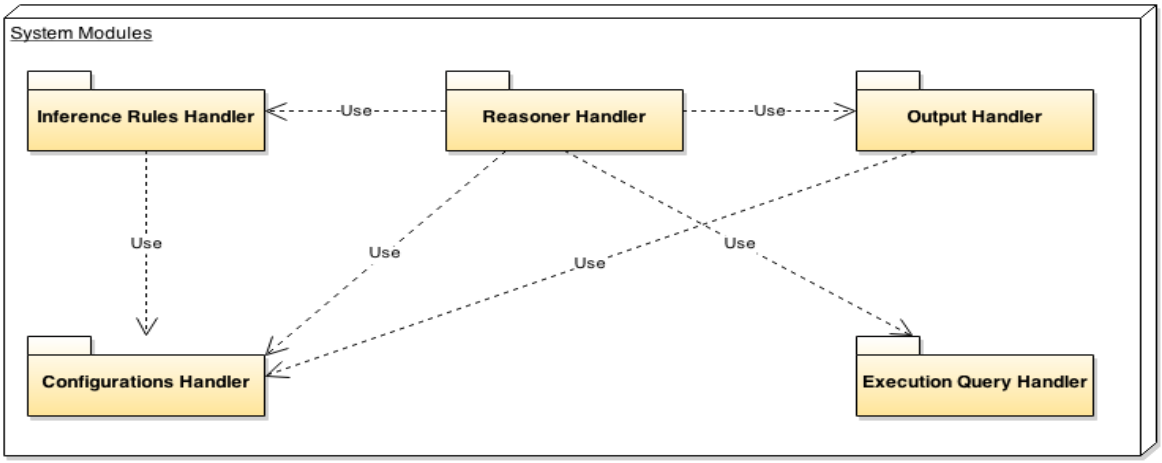

Figure 1. Reasoner Achitecture

- Easy to add new rules using an intuitive language based on the RDF family standard;
- The inference process would point out the list of RDF triples (explicit and inferred ones) used to produce every inferred triple.

In such an approach, the end user is totally aware of the inference process; as a consequence he can evaluate how much it fits his approach to reasoning.

In the rest of the paper, we describe first in Section II the architecture on which the reasoner is based on. Then, in section III, we present the language used to express the inference rules providing some real case examples. In Section IV we show how the reasoner allows the user to understand the logical process over which each new RDF triple has been generated. Finally, in Section V, we present our conclusions.

II. ARCHITECTURE AND IMPLEMENTATION

The architecture of the reasoner Human-readable Ontology Reasoner Unit System (HORUS) is shown in Fig. 1. A modular approach has been adopted to make easy to change any module without modifying the other ones.

First, the configuration parameters are read by the *Configuration Handler* and passed to the *Reasoner Handler*. Then, all inference rules are parsed by the *Inference Rules Handler*. The language in which these rules are written will be discussed in detail in Section III. HORUS does not have any hard-coded inference rule, each rule used by the reasoner is written in the developed language, so a user will be able to see what the reasoner is able to infer, without the need to read its source code. This is a first aspect of configurability.

Once the reasoner is configured, each inference rule is executed by the *Execution Query Handler*, which, in the current implementation, uses SPARQL SELECT [10], taking advantage for any improvement provided by the triple store the reasoner is used with. To avoid any dependency on the specific technology regarding a triple store, HORUS uses the OWL-ART API [11] middle layer which enable an abstraction layer over different RDF triple store technologies. In the current implementation, the reasoner has been tested with these API in conjunction with a Sesame2 implementation [12]. All the inference rules are executed in one or more iterations, until the reasoner is able to infer no further knowledge, or the number of iterations specified during the configuration is achieved.

Finally, the output of the inference process is shown to the user by the *Output Handler*.

III. LANGUAGE

Hereafter, we describe the language defined to specify the *inference rules* and *consistency rules* to be used by the reasoner, that follow a similar syntax while their objective is totally different. The former rule is used to deduce new knowledge (using either already existing or inferred in a previous iteration), the latter does not produce any knowledge, it is used to check if the ontology causes an inconsistency (two or more axioms which contradict each other). In the following, first we explain the syntax adopted for the rules and then we provide some real case rules.

A. Rule syntax and use

The simplified grammar of the language developed for these rule is shown in Fig. 2.

Each rule starts with the word *rule* followed by its type. There exist two possible rule types: inference rule (called *new rule*) and consistency rule (*new consistency rule*).

Successively, the name and an id are provided. The id must be unique and it is used to refer to a specific rule. Then, the list of premises used by the reasoner follows. They check if in the current iteration this rule is able to generate new knowledge. Generally at least two premises are required to

```
parseInferenceRule : (new_rule)+;
new_rule : (rule_info) (premise)+ (filter)* (conclusion)+;
rule_info : 'rule: ' type 'name: ' NAME 'id: ' ID ;
type : 'new rule' | 'new consistency rule';
premise : 'premise: ' triple;
triple : 'subject: ' value 'predicate : ' value 'object: ' value;
filter : 'filter: ' '?' VAR LOGIC_OPERATOR '?' VAR;
value : ('?' VAR) | IRI | BNODE | SINGLEVALUE ;
conclusion : ('conclusion: ' triple ) | ('conclusion: ' 'false' );
```

Figure 2. Simplified rule grammar

 ISBN: 978-1-61208-340-7

have a meaningful rule. Each premise is constituted by three elements: a *subject*, a *predicate* and an *object*. Since in the grammar the reasoner works on RDF datasets, we decided to adopt the same terminology used in RDF. The meaning of the premises is that the reasoner searches the RDF datasets for all the RDF graph which satisfy all the premises of a given rule. Each element of a premise can be one of the following:

- a variable (introduced by the symbol *?* as it is done in the SPARQL grammar);
- an IRI (starting with the symbol <, containing a URI an ending a > or alternatively a prefix followed by a local name);
- a BNODE (using the same syntax in RDF, a *_:* followed by a name), used when we are not interested in the particular value, we just need that it exists, as it is done in SPARQL;
- a SINGLEVALUE, which is a typed literal containing a number.

These premises can be optionally followed by zero or more filter constrains. In the grammar shown in Fig. 2, each filter is represented as being just a comparison between two elements to avoid possible confusion when reading it.

In the real grammar used by HORUS it is possible to define complex comparison using Boolean expression, so it is possible to have a filter which uses the *or* Boolean operator to join several simple comparison. See example later on.

The last part of each rule is the conclusion. When dealing with an inference rule, the conclusion contains one or more triples. These triples are used by the reasoner to know the RDF graph that can be inferred using the current rule. The syntax used by each conclusion is similar to the one used for the premises, because in both cases the reasoner is dealing with RDF triples. The variables used in the conclusion contain the value(s) bound by the reasoner during the inference process. In the retrieve phase of the inference process, the reasoner can retrieve more than one RDF graph which satisfy all the premises of the inference rule. The reasoner then iterates over all the retrieved RDF graphs, and, for each graph, it creates all the RDF triples by using the templates stated in the conclusion section of the rule.

When dealing with a consistency rule, the conclusion can only be *false*. In fact, if the reasoner is able to find at least one RDF graph which satisfies all the premises and the filters, then the ontology contains an inconsistency, so the reasoner generate no new RDF triples; it just needs to save the RDF triples which generated the inconsistency (or at least what has been labeled by the current rule as an inconsistency) to show them to user.

B. Inference and Consistency rules example

To better understand what is possible to achieve by using the previously described grammar, we provide a few real case rules. By first we present the content of a file containing two simple rules; then, we discuss a more complex rule which uses a filter to deal with cardinality restriction regarding the definition of a class; finally, we show an example of a consistency rule

```
type : new rule
name: Transitive
id: 1
premise: subject: ?p predicate: rdf:type object:
        owl:TransitiveProperty
premise: subject: ?a predicate: ?p object: ?b
premise: subject: ?b predicate: ?p object: ?c
conclusion: subject: ?a predicate: ?p object: ?c

type : new rule
name: Symmetric
id: 2
premise: subject: ?p predicate: rdf:type object:
        owl:SymmetricProperty
premise: subject: ?a predicate: ?p object: ?b
conclusion: subject: ?b predicate: ?p object: ?a
```

Figure 3. Two simple Inferemce Rule

1) Simple Inference rules

In the definition of an ontology, it is common to have a property defined as transitive and/or symmetric. The rules used for this particular task are shown in Fig.3. The first one, called *Transitive*, and identified by the id 1, consists of three premises and one conclusion. In the premises, we use the prefix and local name instead of the complete URI (while we suggest to use the complete URI to avoid any confusion). The first premise states that we are interested in all resources which have as one type the value *owl:TransitiveProperty*. We then need to find all the RDF triple of the form *?a ?p ?b* and *?b ?p ?c*, where ?p is bound to a resource (a property in this case) which is *owl:TransitiveProperty* and the variable *?b* of the second premise must bound to the same resources used with the variable *?b* of the third premise. At any iteration, the reasoner searches for any RDF graph which satisfies these three triples and for every graph it applies the conclusion. The reasoner searches for the graph not only in the original ontology, but also in all the inferred triples obtained in the previous application of the rules, so it combines both explicit and inferred knowledge. In this case, there is only one conclusion, *?a ?p ?c*, stating that this triple, where each of the variable is bound to the value found in the query execution phase, should be added to the inferred list of new triples. This triple (or these triples if more than one RDF graph was found) are added to the list of the inferred new triples only if the following two conditions are met:

- The new triples were not already represented in the original ontology;
- The new triple has not been already generated in a previous application of either this or other rules.

When adding a new triple, the reasoner stores the triples which were used in the inferred process, to show them in the log file and in a graph GUI to the user (see Section IV).

The other rule, called *Symmetric* and having id:2 is similar to the first one. Having two premises and one conclusion, it is searching for the resources having type *owl:SymmetricProperty*. It is important to notice that even if two rules share a variable with the same name (in this case the variables *?a , ?p* and *?b*) each variable has the rule itself

type : new rule
name: **ComplexSubClass**
id: 13
premise:subject:?p1predicate:rdfs:subPropertyOf
object: ?p2
premise: subject: ?class1 predicate: owl:equivalentClass
object: ?equiClass1
premise: subject: ?equiClass1 predicate: rdf:type
object: owl:#Restriction
premise: subject: ?equiClass1 predicate: owl:onProperty
object: ?p1
premise: subject: ?equiClass1 predicate: owl:minCardinality
object: ?card1
premise: subject: ?class2 predicate: owl::equivalentClass
object: ?equiClass2
premise: subject: ?equiClass2 predicate: rdf:type
object: owl:Restriction
premise: subject: ?equiClass2 predicate: owl:onProperty
object: ?p2
premise: subject: ?equiClass2 predicate: owl:minCardinality
object: ?card2
filter: ?card1 >= ?card2
conclusion: subject: ?class1 predicate: rdfs:subClassOf
object: ?class2

Figure 4. Complex Inferemce Rule

as its scope, so binding in a rule a variable to a particular value has no effect on the application of another rule.

2) Inference Rule with a Filter

We now describe a more complex rule using the filter after the premises. The idea behind this rule is that if a class *?class1* is defined as equivalent to a class having *minCardinality* on property*?p1* equal to *?card1* AND a second class *?class2* is equivalent to a class having *minCardinality* on property *?p2* equal to *?card2* AND if the value associated to *?card1* is greater or equal to *?card2* AND if the property bound to *?p1* is *subProperty* to the property bound to *?p2*, THEN we can infer *?class1* is a *subClass* of *?class2*.

This complex inference is represented by the rule written in Fig. 4, in which, to infer that a class is a subClass of another class, we need to analyze their restrictions. This rule is constituted by nine premises, one filter and one conclusion. The nine premises can be divided into three sets:

- The first one has just the first premise and regards two properties, *?p1* and *?p2* where *?p1* is a subProperty of *?p1;*
- The second one deals with the definition of a class, *?class1*, and its equivalent class, *?equiClass1*, which has a restriction regarding the *minCardinality*, having value *?card1*, on the property *?p1*. This second set is formed by 4 premises (from premise 2 to premise 5);
- The third and final set is equivalent to the second one, by replacing the variable with *?class2*, *?equiClass2*, *?card2* and *?p2*. Its premises are from premise 6 to premise 9.

The filter is used to check and compare the values of the two cardinalities. In this case the cardinality associated to the first class, *?class1*, is greater or equal to the cardinality associated to the second class, *?class2*.

type : new consistency rule
name: **Same_and_Different**
id: 6
premise: subject: ?a predicate: owl:sameAs object: ?b
premise: subject: ?a predicate: owl:differentFrom object: ?b
conclusion: false

type : new consistency rule
name: **MaxCard_consistency**
id: 12
premise: subject: ?x predicate: owl: maxCardinality
object: "0"^^xsd:nonNegativeInteger
premise: subject: ?x predicate: owl:onProperty object: ?p
premise: subject: ?u predicate: rdf:type object: ?x
premise: subject: ?u predicate: ?p object: ?y
conclusion: false

Figure 5. Consistency Rule

3) Consistency Rule

The three previously described rules highlighted what are the possible inferences that are possible in HORUS. Now, we discuss how to write a rule which is used to check if the ontology, and all the inferred RDF triples, violates any constraint. Fig. 5 contains two consistency rules.

The first one, named *Same_and_Different*, is used to check if there exist two resources (classes in this case) which are defined, or inferred, to be simultaneously *sameAs* and *differentFrom*. In such a case, there is obviously an inconsistency in the vocabulary used in the ontology, because two axioms are mutually exclusive.

The second rule is more complex, it states the presence of an inconsistency if a class *?x* has *maxCardinality* equals to 0 on property *?p* and then in the RDF dataset we have an instance of class *?x*, which has the property *?p*. In this second case as well, the only possible conclusion is *false*.

IV. REASONER USE AND RESULT VISUALIZATION

We now describe how to use HORUS and how to visualize reasoning results. Since it has been developed as a library, it will be invoked inside another tool. Two possible solutions have been developed:

- Inside a simple stand alone Java program;
- Inside a Semantic Turkey Extension.

In the rest of this section, we will describe how use the reasoner into Semantic Turkey framework.

Semantic Turkey [13][14] supports an ontology editor developed as an extension of the popular web browser Firefox [15] with a client/server architecture. One main feature is its extendibility, achieved by developing new extensions by using both Java plugin framework OSGi Felix [16] and the Firefox extension mechanism. Each Semantic Turkey extension consists of two part:

- A Java implementation, which extends the server side and is written completely in Java;
- A Firefox extension, written in JavaScript and XUL (client side) and responsible for the interaction with the user taking advantages of the Firefox GUI.

 ISBN: 978-1-61208-340-7

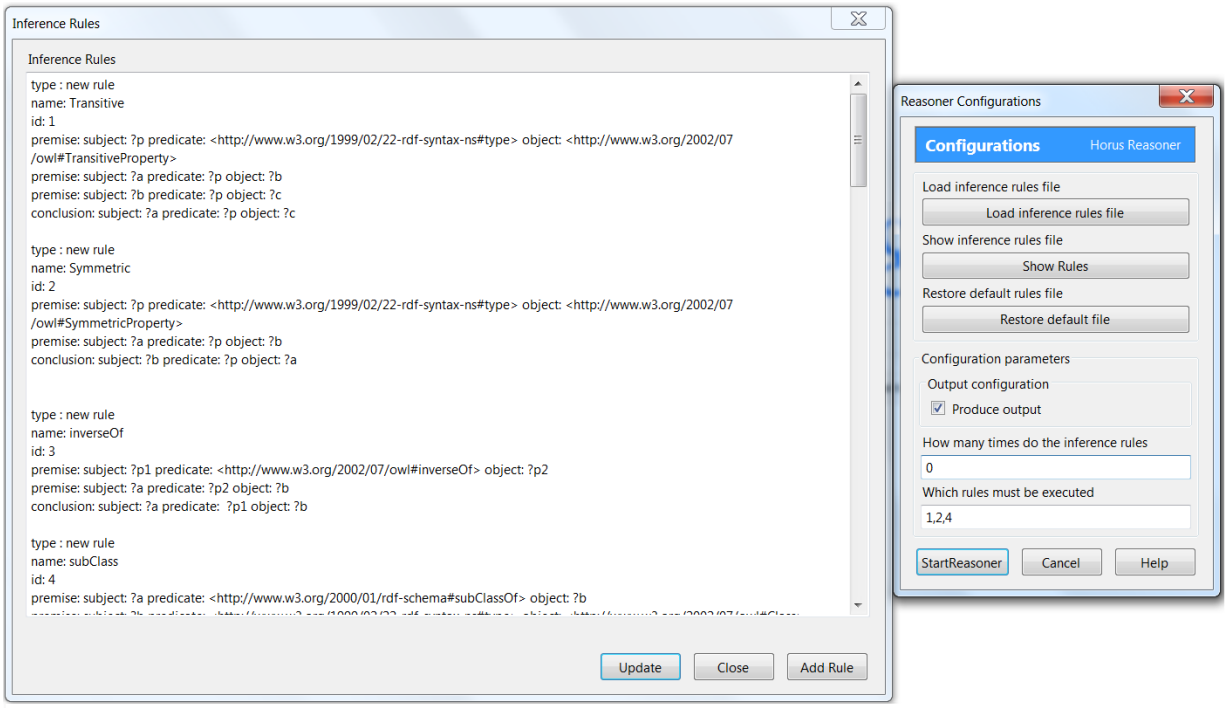

Figure 6. HORUS inside Semantic Turkey

HORUS was placed inside a Semantic Turkey extension, which can be download from [17] with its source code and the source code of the reasoner as well. Since the reasoner needs at least one inference rule to work, in the downloadable package, a file containing several working and tested inference rules is provided. In what follows, an evidence of configurable property oh HORUS is described.

When the reasoner is executed inside Semantic Turkey, the user has the possibility to decide which file containing the inference/consistency rule to load, which rule among them to use, how many iterations the reasoner should do (if it select 0 then the reasoner will stop only when no new knowledge can be inferred). An example of the presented GUI can be seen in Fig. 6. It is possible to write new rules using a dedicated GUI, which in the next release of the tool will provide a better assistance to the user for this task.

Once the user has selected which rule file to load and which rules to use, he can launch the reasoning process on the ontology currently managed by Semantic Turkey.

At the end of the reasoning, the inferred RDF triples are added in the current ontology in a different graph, which can be deleted at any moment by the user for several reasons (for example the ontology has changed and the inferred triples are no longer valid, because they cannot be derived from the new ontology).

The user is able to check all the inferred knowledge in two complementary ways:

- In the logger file, containing all inferred RDF triples with all the knowledge used to generate them and the name of the rule used in the process;
- in a graph, where each node is an RDF triple (explicit or inferred) and each link states which RDF triples were used to generate other triples.

An example of the result graph can be seen in Fig. 7. In this case, we have execute HORUS on a small ontology dealing with some geographical information about Lazio, a region in Italy. At the center of the graph, for example, we have the triple *Roma locatedIn Italia* generated using three (explicit) triples:

- LocatedIn type TransitiveProperty;
- Roma locatedIn Lazio;
- Lazio locatedIn Italia.

On the left side we have another triple, *Ariccia locatedIn Italia*, which has been inferred from the explicit:

- LocatedIn type TransitiveProperty;
- Ariccia locatedIn Roma.

and the previous inferred:

- Roma locatedIn Italia.

Finally, on the left side of the GUI in Fig. 7, we see a series of button that can be used to switch between the graph representation or the text one (the logging file) and to delete all the inferred triple (by deleting the RDF graph in the ontology in which they are stored). Using the GUI interface the user is also able to filter the results, to concentrate its attention to just a particular RDF inferred triple and the knowledge that was used to produce it.

The consistency rules are not shown in the graph representation, they are present only in the logging file.

V. CONCLUSION

In this article, we have presented a first implementation of HORUS, a new reasoner whose main features are:

- Possibility to write new inference and consistency rules by using an intuitive language based on some concepts of RDF standard and SPARQL filter;

 ISBN: 978-1-61208-340-7

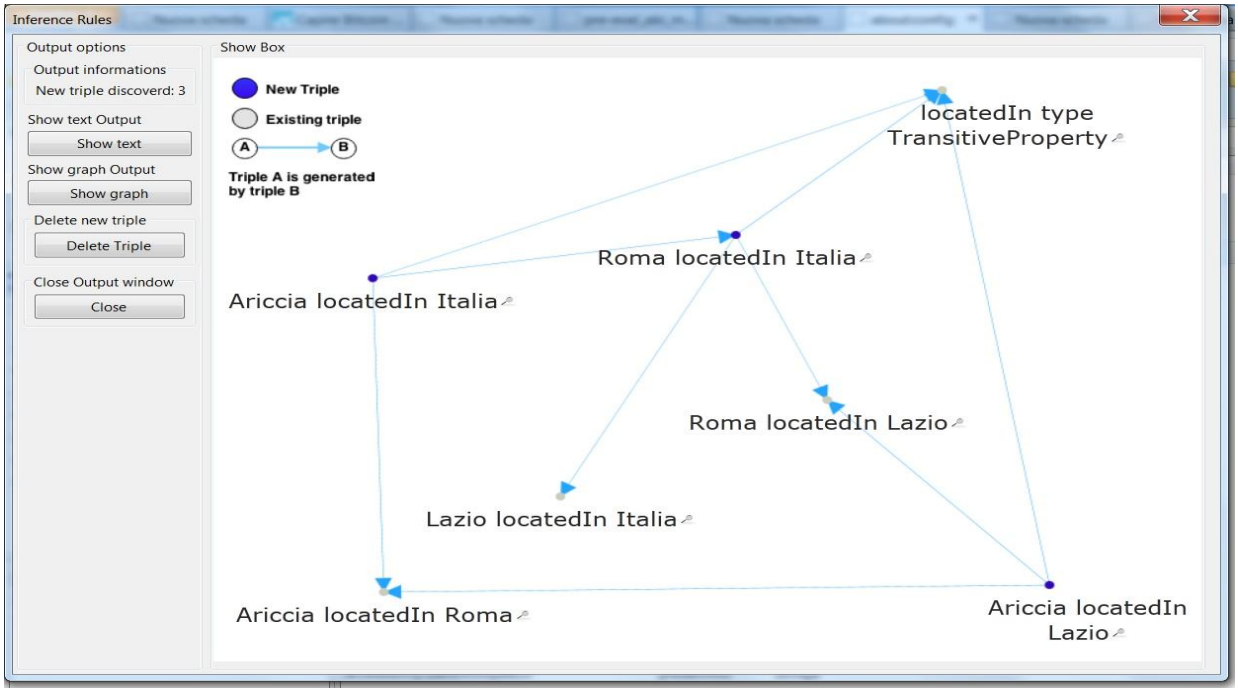

Figure 7. Inferred RDF triples in a graph

- Being aware of why each new triple was inferred by consulting a graphical representation or by reading a logging file containing all the motivations for each decision taken by the logger.

Possible applications are foreseen in context as:

- Educational use, to teach ontologies and inferences;
- Understanding why inferred triples were generated;
- Understanding which axioms should be changed or deleted in the ontology to prevent an undesired inference.

In fact, knowledge representation and reasoning techniques can be used for modeling background knowledge (e.g., in the form of ontologies) and to reason over them for logic-based verification.

REFERENCES

[1] W3C, Resource Description Framework (RDF), 2004. [Online]. Available: http://www.w3.org/RDF/ [retrieved: Apr, 2014]

[2] Description Logic Reasoners. [Online]. http://www.cs.man.ac.uk/~sattler/reasoners.html [retrieved: Apr, 2014]

[3] M. Horridge, J. Bauer, B. Parsia, and U. Sattler, "Understanding Entailments in OWL," in Fifth OWLED Workshop on OWL, Karlsruhe, Germany, 2008.

[4] J. Gennari, et al. "The evolution of Protégé-2000: An environment for knowledge-based systems development," International Journal of Human-Computer Studies, 2003, vol. 58, n. 1, pp. 89–123.

[5] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," Web Semantics: science, services and agents on the World Wide Web, 2007, vol. 5, no. 2, pp. 51-53.

[6] Pellet FAQ: Single Page Version [Online]. http://clarkparsia.com/pellet/faq/single-page#rules [retrieved: Apr, 2014]

[7] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. [Online]. http://www.w3.org/Submission/SWRL/ [retrieved: Apr, 2014]

[8] D. Tsarkov and I. Horrocks, "FaCT++ Description Logic Reasoner: System Description," in Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, WA, Springer-Verlag, 2006, pp. 292-297.

[9] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe, "The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs," in Proceedings of the 2012 Haskell Symposium (Haskell '12), New York, NY, USA, 2012, pp. 1-12.

[10] SPARQL Query Language for RDF. [Online]. http://www.w3.org/TR/rdf-sparql-query/ [retrieved: Apr, 2014]

[11] Official OWL ART API website. [Online]. http://art.uniroma2.it/owlart/ [retrieved: Apr, 2014]

[12] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in The Semantic Web - ISWC 2002: First International Semantic Web Conference, Sardinia, Italy, 2002, pp. 54-68.

[13] M. T. Pazienza, N. Scarpato, A. Stellato, and A. Turbati, "Semantic Turkey: A Browser-Integrated Environment for Knowledge Acquisition and Management," Semantic Web, Jan 2012, vol. III, no. 3, pp. 279-292.

[14] Semantic Turkey Homepage. [Online]. http://semanticturkey.uniroma2.it [retrieved: Apr, 2014]

[15] Firefox Homepage. [Online]. http://www.mozilla.org/en-US/firefox/new/ [retrieved: Apr, 2014]

[16] Apache Felix Homepage. [Online]. http://felix.apache.org/ [retrieved: Apr, 2014]

[17] HORUS Homepage. [Online]. https://code.google.com/p/reasoner [retrieved: Apr, 2014]

 ISBN: 978-1-61208-340-7