Querying and Visualizing the Semantic Web

Georgios Meditskos[1], Efstratios Kontopoulos[1], Nick Bassiliades[1]

[1] Department of Informatics, Aristotle University of Thessaloniki, GR-54124 Thessaloniki, Greece, {gmeditsk, skontopo, nbassili}@csd.auth.gr

The amount of unstructured information on the Web has reached a critical level, suggesting the need for the Semantic Web (SW). The SW expresses an initiative to improve the World Wide Web, by adding semantic content to the provided information, thus making the data accessible not only to humans but to machines as well. The fulfilment of this goal requires the contribution of a variety of research areas and tools: reasoning techniques that give the opportunity to agents/machines to act autonomously on the environment of the Web in order to fulfil the users' needs and tools for assisting the end users in expressing these needs and comprehending the derived results. This paper presents two systems, namely O-DEVICE and VDR-DEVICE, developed by the Intelligent Systems and Knowledge Processing (ISKP) group[1] at the Department of Informatics, Aristotle University of Thessaloniki, Greece. These two systems are designed to function in the SW environment. O-DEVICE is a system for querying and inferencing about ontologies expressed in the OWL Lite sublanguage and VDR-DEVICE is a visual integrated development environment for developing, using and visualizing defeasible logic rule bases on top of RDF ontologies.

Keywords

Semantic Web, Reasoning System, Visualization Techniques, RuleML, Defeasible Logic

1. Introduction

The Semantic Web (SW) [1] constitutes an effort to improve the current Web, by targeting at the meaning rather than the presentation of information. This way the content of the Web is made accessible not only to humans, as it is today, but to machines/agents as well, with the latter being now able to understand the available information on the Web and act on behalf of users, in order to achieve their goals in an automated and accurate way.

To describe information appropriately, knowledge representation languages based on XML have been proposed, such as RDF [2] and OWL [3]. Both languages are used to annotate the information in a formal and explicit way by defining ontologies, using classes, properties and instances of classes. The difference between the two languages is the degree of expressiveness they offer. Thus, while RDF is capable of defining only subclass and subproperty relationships between classes and properties respectively, OWL goes a step further by offering a higher degree of expressiveness. Having been built on top of RDF, OWL introduces more relationships between classes, properties and instances, allowing the use of property constraints and Boolean operators (union, intersection, etc) in class definitions.

To fulfil the goal of the SW - a Web understandable by machines - the annotation of information is not enough by itself. There also exists the need for a variety of appropriate tools such as reasoners, graphical editors, visualisation tools and environments that successfully integrate the various technologies and languages involved in such applications. Reasoners are systems able to process SW information and answer queries over that information, so they play a significant role, since they are also able to extract new information that is not stated explicitly, based on the formal semantics of the language used for annotation. The rest of the tools mentioned above are equally important, since they

[1] http://iskp.csd.auth.gr/

can greatly assist the end user in exploiting the SW to its full extend as well as for gradually increasing the user trust towards the SW.

In this paper we present two systems: (a) O-DEVICE, a production rule-based system for inferencing and querying ontologies expressed in the OWL Lite language (the simplest sublanguage of OWL), and (b) VDR-DEVICE, a visual integrated development environment for developing, using and visualizing defeasible logic rule bases (rule bases based on defeasible reasoning – an approach to reasoning with incomplete, changing or conflicting information) on top of RDF ontologies. Both systems were developed by the Intelligent Systems and Knowledge Processing (ISKP) group at the Programming Languages and Software Engineering (PLaSE) Laboratory of the Department of Informatics, Aristotle University of Thessaloniki, Greece.

O-DEVICE utilizes an existing production rule system, named CLIPS [4], taking advantage of its efficiency and speed and augments it with an OWL-to-objects mapping mechanism and a deductive query language in order to handle OWL semantics. The system performs inference by following an approach we call Dynamic Rule Generation and it is able to answer queries over the instances of the knowledge base.

VDR-DEVICE, on the other hand, integrates in a user-friendly graphical shell, a defeasible reasoning system that processes RDF data and RDF Schema ontologies, a visual RuleML-compliant rule editor and graph-generating tools for visualizing defeasible logic rule bases. The system supports the multiple rule types of defeasible logic, as well as priorities among rules. It also supports two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals.

Both systems are based on a previous implementation, called R-DEVICE [5], which has been developed by our group too. R-DEVICE is a deductive object-oriented knowledge base system for reasoning over RDF metadata. It is based on an OO RDF data model, different than the established triple-based model, which maps resources to objects and encapsulates properties inside resource objects, as traditional OO attributes. R-DEVICE features a powerful deductive rule language [5] which is able to draw and materialize inference both on the RDF schema and data. The rule language includes features such as ground and generalized path expressions, stratified negation, aggregate, grouping, and sorting functions. The syntax of R-DEVICE rules follows the OPS5/CLIPS paradigm. Furthermore, an XML syntax is provided that extends RuleML [6] and especially the version that supports OO features and negation-as-failure.

The rest of the paper is organized as follows: Section 2 presents the O-DEVICE system more extensively, while section 3 displays the key-features and functionality points of the VDR-DEVICE system. Finally, section 4 concludes this paper and poses directions for future work.

2. O-DEVICE: Inferencing about and Querying OWL Ontologies

The ability to infer new information is a critical characteristic of every reasoning system and defines its completeness and soundness. Unfortunately, there is a tradeoff between scalability, in terms of processing and querying time, and reasoning capabilities. For example, it is almost infeasible to build a complete reasoning system for OWL Full because the great degree of expressiveness does not offer computational guarantees. Thus, most of the reasoning systems aim at the two sublanguages of OWL, OWL Lite and OWL DL that demand fewer reasoning capabilities, especially OWL Lite. Furthermore, the majority of realistic web applications usually involve large number of instances, so a reasoning system should be able to handle efficiently large ABoxes (queries over instances).

The O-DEVICE ([7], [8]) is a system that inferences over (on top of) OWL documents. It exploits the advantages of the object-oriented programming model by transforming OWL ontologies into classes, properties and objects of the OO programming language provided within CLIPS, called COOL.

While most rule-based inference engines utilize rules that operate at the level of triples, in our approach we consume triples to build an object-oriented schema out of an ontology and to materialize OWL instances as objects that encapsulate their properties into their slots. We utilize production rules to infer knowledge based on the ontology and perform appropriate actions in order to materialize the inferred knowledge on the schema and on the knowledge base. The system also supports a deductive rule language which is used to express queries on the OWL instances. The advantage of this object-oriented approach is that knowledge derived from ontologies is no more scattered in the form of triples across the memory. Instead, rules match a considerable less number of objects (than triples). To make the matching procedure even faster, we introduce a *Dynamic Rule Generation* approach that generates *domain dependent* inference rules that have simpler conditions and thus, faster activation time.

2.1 System Description

The system consists of three basic modules: the *OWL Translator*, the *Query Translator* and the *Result Extractor* (Figure 1). All modules are connected to the CLIPS Production Rule System. The next paragraphs briefly describe each module functionality and role in the overall architecture.

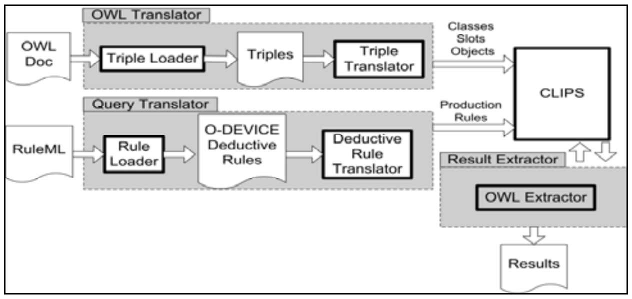


Figure 1 O-DEVICE architecture

OWL Translator. This module is responsible for transforming an OWL document into the object-oriented schema of CLIPS Object-Oriented Language (COOL). It consists of the *Triple Loader* and the *Triple Translator*. The *Triple Loader* loads an OWL document into the system. The document may be stored either locally or remotely, i.e. accessed through a URL. With the aid of the *ARP Parser* [9] the document is transformed into a set of triples which are loaded into main memory and are forwarded to the *Triple Translator*. The latter consists of a set of CLIPS production rules that transforms OWL constructs into COOL object-oriented schema of classes, slots and objects.

Query Translator and Result Extractor. It is responsible for translating incoming queries, which are expressed as deductive rules, into CLIPS production rules. Queries can be expressed either by using a RuleML-like syntax and then transform them into the native CLIPS-like rule language using the *Rule Loader,* or directly into the native syntax. In both ways, the deductive rules are finally passed into *Deductive Rule Translator* which compiles them into CLIPS production rules.

Result Extractor. It is responsible for extracting the results of the executed queries from CLIPS and exporting them into a file using OWL/RDF syntax.

2.2 Building the Object-Oriented Schema

An OWL/RDF document consists of a set of statements in the form of <*subject predicate object*>. Each such statement (triple) denotes the relation that exists between the *subject* and the *object* through the *predicate*. The system reads these triples and materializes the explicit information derived from them according to OWL semantics, using rules and a predefined object-oriented schema in COOL of the OWL built-in classes and properties [10].

Classes. User-defined classes of the ontology are instances of *owl:Class*. For each OWL class such an object (called *meta-object*) is created and the system uses the information stored in them in order to build the actual CLIPS classes (defclass constructs).

Appropriate rules track such objects and build the corresponding class hierarchy (which may need redefinition) based on the values of *rdfs:subClassOf*, *owl:intersectionOf* and *owl:equivalentClass* constructs. If there are no values for these slots, classes become direct subclasses of *owl:Thing*.

Property Instances. These instances define properties in the ontology. Properties could be either instances of *owl:DatatypeProperty* or *owl:ObjectProperty* classes (including their subclasses) and the system generates the objects of the corresponding classes. Rules track such objects and based on the *rdfs:domain* and *rdfs:range* values, create the corresponding slots in the classes. During this procedure, classes may need redefinition if they have already been created.

User-defined class instances. These are the actual objects of user-defined classes in the object-oriented schema. By the time these objects are created, system can populate their slots (user-defined properties) with the values from the ontology instances.

2.3 Extended OWL Semantic Transformations with a Dynamic Rule Generation Technique

In addition to the basic transformations that the system applies at load time in order to build an initial object-oriented infrastructure, it also utilizes production rules to infer information that stems from the semantics of OWL. Since, in our approach, the information is represented using objects and slots, production rules require several joins between the objects of the knowledge base. Thus, the more data we have, the more joins are required, and experiments have shown that increasing number of joins results in an almost exponential increase of load time.

The above observation, led us to adopt an approach based on *Dynamic Rule Generation*. The system generates inference rules based every time on the specific characteristics of the ontology. In that way, rules have only a single condition in their head, avoiding multiple joins and check the minimum possible number of objects. The example that follows demonstrates the advantage of this approach, describing the way that our system handles transitive properties.

In OWL, when a property P is transitive and the pairs (x, y) and (y, z) are instances of P, then it can be inferred that the pair (x, z) is also an instance of P. A general rule for all transitive properties would require three condition elements in its head: one for the transitive property and two more for the objects that participate in the transitive closure. Below we show a simplified rule that implements this.

```
(defrule transitive-property
   (object (is-a owl:TransitiveProperty)(name ?property)
   (rdfs:domain ?domain))
   (object (is-a ?domain)(name ?obj1)(?property $?values1))
   (object (is-a ?domain)(name ?obj2)(?property $?values2))
   (test (belongs ?obj2 $?values1))
=>
  (put-unique-values $?values2 $?values1))
```

Experiments have shown that this kind of rules, i.e rules with more than two condition elements, require much time to be activated, especially when we deal with many objects where the number of joins increases dramatically.

Following the *Dynamic Rule Generation* approach we described, for every transitive property, we generate a rule that grounds as many variables as possible. These rules have limited complexity since they have only one object pattern to match with the minimum number of variables. Consider the following example of a transitive property named *subRegionOf*:

```
<owl:Class rdf:ID="Region"/>
<owl:TransitiveProperty rdf:ID="subRegionOf">
   <rdfs:domain rdf:resource="#Region"/>
   <rdfs:range rdf:resource="#Region"/>
</owl:TransitiveProperty>
<Region rdf:ID="region1" >
   <subRegionOf rdf:resource="#region2"/>
</Region>
```

Region1 is a *subRegionOf region2* and *region2* is a *subRegionOf region3*. Because *subRegionOf* is a transitive property, the system must infer that *region1* is also a *subRegionOf region3*. The resulting rule for the above property following the *Dynamic Rule Generation* approach can be seen below.

```
(defrule gen1
   (object (is-a Region)(name ?obj1)
   (subRegionOf $? ?obj2 $?))
   (test (transitive-closure ?obj1 ?obj2 subRegionOf)
=>
    (put-unique-values ?obj2 ?obj1))
```

This rule checks every instance value of the *subRegionOf* property of the instance *?name* and in the body of the rule computes the transitive closure. In that way, only instances of the class *Region* and its subclasses are checked, without performing any join in order to calculate the transitive closure, speeding-up rule activation time.

2.4 The Deductive Rule Language of O-DEVICE

The deductive rule language of O-DEVICE supports inferencing over OWL instances represented as objects and defines materialized views over them, possibly incrementally maintained. The conclusions of deductive rules represent derived classes, i.e. classes whose objects are generated by evaluating these rules over the current set of objects. Furthermore, the language supports recursion, stratified negation, path expressions over the objects, generalized path expressions (i.e. path expressions with an unknown number of intermediate steps), derived and aggregate attributes [5]. Each deductive rule in O-DEVICE is implemented as a CLIPS production rule that inserts a derived object when the condition of the deductive rule is satisfied.

The following example shows a rule that retrieves the names of all *Woman* instances that have a value less than 22 in the age property by deriving instances of class *young-woman* with the value *?fname* in the *fname* property:

```
(deductiverule young-women
   (test:Woman (test:age ?x&:(< ?x 22)) (test:fname ?fname))
=>
   (young-woman (fname ?fname)))
```

3. VDR-DEVICE: Developing and Visualizing Defeasible Logic Rule Bases

Defeasible reasoning [11], a member of the non-monotonic reasoning family, represents a simple rule-based approach to reasoning with incomplete, changing and conflicting information. The main advantages of this approach are enhanced representational capabilities coupled with low computational complexity.

Defeasible reasoning can represent facts, rules and priorities and conflicts among rules. Nevertheless, although defeasible logic is certainly a very promising reasoning technology, the development of rule-based applications for the SW can be greatly compromised by two factors. First, the RuleML syntax of defeasible logic, which is briefly exhibited in a next section, is certainly too complicated for an end-user language, creating the need for user-friendly authoring tools. Furthermore, the solid mathematical formulation that forms the basis of defeasible reasoning may seem too complicated. In this case a graphical trace and an explanation mechanism would also be very beneficial.

VDR-DEVICE is a visual integrated development environment for developing, using and visualizing defeasible logic rule bases on top of RDF ontologies. The system integrates in a user-friendly graphical shell, a defeasible reasoning system that processes RDF data and RDF Schema ontologies, a visual RuleML-compliant rule editor and graph-generating tools for visualizing defeasible logic rule bases.

3.1 Defeasible Reasoning – Basic Principles

The main building block of defeasible reasoning is the *defeasible theory D* (i.e. a knowledge base or a program in defeasible logic), which consists of three basic ingredients: a set of facts (F), a set of rules (R) and a superiority relationship (>). Therefore, D can be represented by the triple (F, R, >).

In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters.

(a) *Strict rules* are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, then so is the conclusion. An example of a strict rule is: "*Penguins are birds*", which would become: $\texttt{r}_1$`: penguin(X) → bird(X)`.

(b) *Defeasible rules*, contrary to strict rules, can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. Examples of defeasible rules are $\texttt{r}_2$`: bird(X) ⇒ flies(X)`, which reads as: "*Birds typically fly*" and $\texttt{r}_3$`: penguin(X) ⇒ ¬flies(X)`, namely: "*Penguins typically do not fly*".

(c) *Defeaters*, denoted by *A ~> p*, cannot actively support conclusions, but can only prevent some of them. In other words, they are used to defeat some defeasible conclusions by producing evidence to the contrary. An example of such a defeater is: $\texttt{r}_4$`: heavy(X) ~> ¬flies(X)`, which reads as: "*Heavy things cannot fly*". This defeater can defeat the (defeasible) rule $\texttt{r}_2$ mentioned above.

Finally, the *superiority relationship* among the rule set R is an acyclic relation > on R, that is, the transitive closure of > is irreflexive. Superiority relationships are used, in order to resolve conflicts among rules. For example, given the defeasible rules $\texttt{r}_2$ and $\texttt{r}_3$, no conclusive decision can be made about whether a penguin can fly or not, because rules $\texttt{r}_2$ and $\texttt{r}_3$ contradict each other. But if the superiority relationship $\texttt{r}_3 > \texttt{r}_2$ is introduced, then $\texttt{r}_3$ overrides $\texttt{r}_2$ and we can indeed conclude that the penguin cannot fly. In this case rule $\texttt{r}_3$ is called *superior* to $\texttt{r}_2$ and $\texttt{r}_2$ *inferior* to $\texttt{r}_3$.

3.2 Rule Editor – Design and Functionality

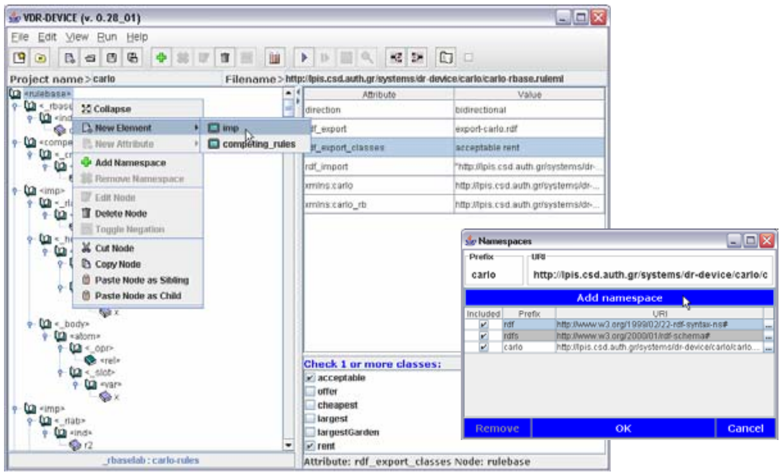
VDR-DEVICE supports a RuleML-compatible [6] syntax, which is the main standardization effort for rules on the SW. Figure 2 displays a fragment of the VDR-DEVICE RuleML-compatible syntax that represents rule $\texttt{r}_1$ mentioned in the previous section.

```
<imp>
      <_rlab ruleID="r3" ruletype="defeasiblerule" superior="r2">
            <ind>r3</ind>
      </_rlab>
      <_head>
            <neg>
                  <atom>
                        <_opr> <rel>flies</rel> </_opr>
                        <_slot name="name"> <var>X</var> </_slot>
                  </atom>
            </neg>
      </_head>
      <_body>
            <atom>
                  <_opr> <rel>penguin</rel> </_opr>
                  <_slot name="name"> <var>X</var> </_slot>
            </atom>
      </_body>
</imp>
```

Figure 2 A strict rule, written in the RuleML-compatible language of VDR-DEVICE.

Figure 2 clearly demonstrates that the RuleML syntax can be too complex for the end user. Thus, the need for authoring tools that assist end-users in writing and expressing rules is imperative. VDR-DEVICE is equipped with DRREd (Figure 3), a visual rule editor that aims at enhancing user-friendliness and efficiency during the development of VDR-DEVICE RuleML documents [12]. Its implementation is oriented towards simplicity of use and familiarity of interface. And its key features include: (a) functional flexibility - program utilities can be triggered via a variety of overhead menu actions, keyboard shortcuts or popup menus, (b) improved development speed - rule bases can be developed in just a few steps and (c) powerful safety mechanisms – the correct syntax is ensured and the user is protected from syntactic or RDF Schema related semantic errors.

The rule base is displayed in XML-tree format, one of the most intuitive means of displaying RuleML-like syntax because of its hierarchical nature. The user can traverse the tree and can add to or remove elements from the tree. However, since each rule base is backed by a DTD document, potential addition or removal of tree elements has to obey to the DTD limitations. Therefore, the rule editor allows a limited number of operations performed on each element, according to the element's meaning within the rule tree.



Attribute	Value
direction	bidirectional
f_export	export-carlo.rdf
f_export_classes	acceptable rent
rdf_import	"http://lpis.csd.auth.gr/systems/dr-...
xmlns:carlo	http://lpis.csd.auth.gr/systems/dr-...
xmlns:carlo_rb	http://lpis.csd.auth.gr/systems/dr-...

Check 1 or more classes:
- [✓] acceptable
- [ ] offer
- [ ] cheapest
- [ ] largest
- [ ] largestGarden
- [✓] rent

Attribute: rdf_export_classes Node: rulebase

Prefix	URI
carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/c

Included	Prefix	URI
✓	rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
✓	rdfs	http://www.w3.org/2000/01/rdf-schema#
✓	carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo...

Figure 3 The graphical rule editor and the namespace dialog window.

By selecting an element from the tree, the corresponding attributes are displayed each time. The user can also perform editing functions on the attributes, by altering the value for each one of them. However, the values that the user can insert are obviously limited by the chosen attribute each time.

3.3 The Reasoning System - Architecture and Functionality

The core reasoning system of VDR-DEVICE is DR-DEVICE [13] and consists of two primary components (Figure 4): The *RDF loader/translator* and the *rule loader/translator*. The user can either develop a rule base with the help of the rule editor described previously, or he/she can load an already existing one, probably developed manually. The rule base contains: (a) a set of rules, (b) the URL(s) of the RDF input document(s), which is forwarded to the RDF loader, (c) the names of the derived classes to be exported as results and (d) the name of the RDF output document.

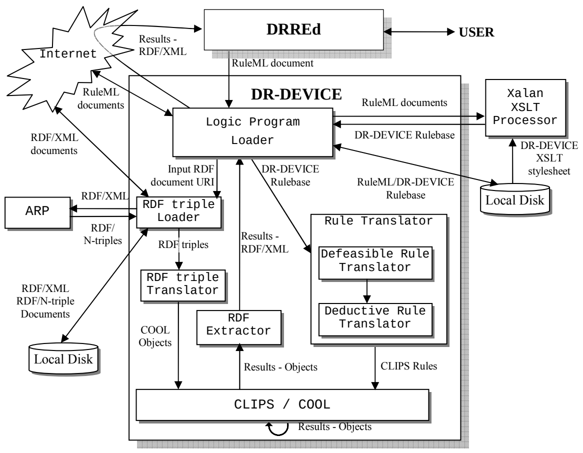


Figure 4 The architecture of the reasoning system

The rule base is then submitted to the *rule loader* which transforms it into the native CLIPS-like syntax through an XSLT stylesheet and the resulting program is then forwarded to the *rule translator*, where the defeasible logic rules are compiled into a set of CLIPS production rules. This is a two-step process: First, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic deductive rule language, using the translation scheme described in [13]. Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in [5].

Meanwhile, the *RDF loader* downloads the input RDF documents, including their schemas, and translates RDF descriptions into CLIPS objects, according to the RDF-to-object translation scheme in [5], which was also described in section 2.

The inference engine of CLIPS performs the reasoning by running the production rules and generates the objects that constitute the result of the initial rule program. The compilation phase guarantees correctness of the reasoning process according to the operational semantics of defeasible logic. Finally, the result-objects are exported to the user as an RDF/XML document through the RDF extractor. The RDF document includes the instances of the exported derived classes, which have been proved.

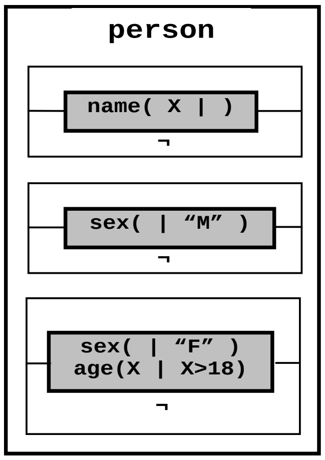
3.4 Visualizing a Defeasible Logic Rule Base

DRREd is equipped with a utility that allows the creation of a directed rule graph from the defeasible rule base developed by the editor [14]. This way, users are offered an extra means of visualizing the rule base, besides XML-tree format described above and, thus, possess a better aspect of the rule base displayed and the inference results produced.

More specifically, in order for the desired graph to be constructed, certain rule base elements have to be collected, following the path described in the following paragraphs. For every class in the rule base (i.e. classes that lie at rule bodies and heads) a *class box* is constructed, which is simply a container. The class boxes are populated with one or more *class patterns* during the development of the rule base. For each atomic formula inside a rule head or body, a new class pattern is created and is associated with the corresponding class box. In practice, class patterns express conditions on instances of the specific class.

Each class pattern is visually represented by a rectangle, separated in two adjacent "*atomic formula boxes*", with the upper one of them representing the positive atomic formula and the lower one representing the negative atomic formula. This way, the atomic formulas are depicted together clearly and separately, maintaining their independence.

Class patterns are populated with one or more *slot patterns*. Each slot pattern consists of a slot name and, optionally, a variable and a list of value constraints. The variable is used in order for the slot value to be unified, with the latter having to satisfy the list of constraints. In other words, slot patterns represent conditions on slots (or class properties).



```
<atom>
  <_opr>
    <rel href="person"/>
  </_opr>
  <_slot name="sex">
    <ind>"F"</ind>
  </_slot>
  <_slot name="age">
    <_and>
      <var>x</var>
      <function_call
       name="&gt;">
        <var>x</var>
        <ind>18</ind>
      </function_call>
    </_and>
  </_slot>
</atom>
```

Figure 5 A class box example and a code fragment for the third class pattern

An example of all the above can be seen in Figure 5. The figure illustrates a class box that contains three class patterns applied on the *person* class and a code fragment matching the third class pattern, written in the RuleML-like syntax of VDR-DEVICE. The first two class patterns contain one slot pattern each, while the third one contains two slot patterns. As can be observed, the argument list of each slot pattern is divided into two parts, separated by ”|”; on the left all the variables are placed and on the right all the corresponding expressions and conditions, regarding the variables on the left. In the case of constant values, only the right-hand side is utilized; thus, the second class pattern of the box in Figure 5, for example, refers to all the *male* persons. This way the content of the slot arguments is clearly depicted and easily comprehended.

A plan has to be drawn, concerning the location of the elements in the graph drawing panel of VDR-DEVICE. For this affair, a variation of the rule stratification algorithm found in [14] was implemented. According to the algorithm, the graph elements are placed in *strata*; each stratum is considered as a *column* in the graph drawing panel of the system. Thus, the first stratum is mapped to the first column on the left, the second stratum to the column on the right of the first one and so on. The algorithm aims at giving a left-to-right orientation to the flow of information in the graph; i.e. the arcs in the digraph are directed from left to right, making the derived graph less complex, more comprehensible and easily readable.

The two aspects of the rule base, namely the XML-tree and the directed graph are interrelated, meaning that traversal and alterations in one will also be reflected in the other. So, if for example the user focuses on a specific element in the tree and then switches to the digraph view, the corresponding element in the digraph will also be selected and the data relevant to it displayed.

Figure 6 displays the output of the VDR-DEVICE rule graph drawing module, when a small defeasible logic rule base is loaded. The specific rule base includes four rules (rules r_1-r_4), all of them are defeasible and the first one “lands” on a positive conclusion (the upper atomic formula box of the head class pattern), while the other three “land” on a negative one (the lower atomic formula box of the head). Furthermore, there exist one base and one derived class. The total number of strata in this case is three (i.e. three columns).

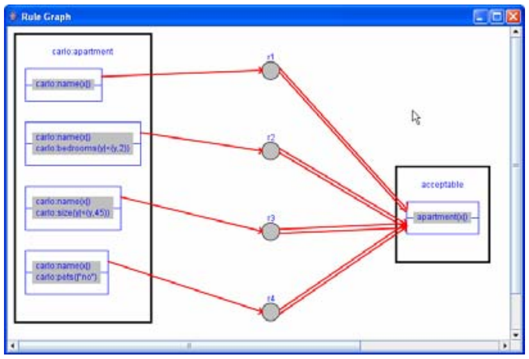


Figure 6 The rule graph drawing module of DRREd

4. Conclusions and Future Work

The Semantic Web is a very promising technology that aims at enhancing the current Web with metadata, thus making its content accessible to machines besides humans. In this paper we presented two systems, O-DEVICE and VDR-DEVICE, developed specifically for the environment of the Semantic Web. The former inferences over OWL metadata, exploiting the advantages of the object-oriented programming model by transforming OWL ontologies into CLIPS classes, properties and objects. The latter constitutes an integrated environment for the development and visualization as well as for the reasoning over defeasible logic rule bases on top of RDF ontologies. Both systems, each in its own way, attempt to narrow the gap between the SW and the end user, by providing means that assist in exploiting the potential of the SW to its full extend.

For the future, we intend to augment the O-DEVICE semantic translation procedure in order to handle more OWL constructs, such as *owl:sameAs*. Furthermore, we plan to extend the system to be able to handle ontologies expressed in the OWL DL language, a more expressive sublanguage of OWL. Our final goal is to develop a Semantic Web Service composition system, where services will be described using OWL-S while the composed services will be described using logic rules.

As for VDR-DEVICE, it is planned to be extended soon with a graphical RDF ontology and data editor that will comply with the user-interface of the RuleML editor. Furthermore, we plan to delve deeper into the proof layer of the Semantic Web architecture by enhancing further the rule representation utility demonstrated with rule execution tracing, explanation, proof exchange in an XML or RDF format, proof visualization and validation, etc. These facilities would be useful e.g. for automating proof exchange and trust among SW agents and for eventually increasing the trust of users towards the SW.

The ultimate goal is naturally to merge the two systems into one, by having a single visual environment for developing and deploying rule bases of varying expressiveness and functionality in the SW for a variety of ontology and metadata languages. Further development for the integrated tool would be the persistent storage of metadata in a SW repository, such as Sesame [16] or ICS-FORTH RDF Suite [17] and its deployment as a Semantic Web service.

References

1. Berners-Lee T, Hendler J, Lassila O. The Semantic Web. *Scientific American*, 2001; 284(5), 34-43.

2. Resource Description Framework, http://www.w3.org/RDF/

3. Web Ontology Language (OWL), http://www.w3.org/2004/OWL/

4. CLIPS 6.23 Basic Programming Guide, http://www.ghg.net/clips

5. Bassiliades N, Vlahavas I. R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata. *Int. Journal on Semantic Web and Information Systems*, 2(2), pp. 24-90, 2006.

6. Boley H, Tabet S, Wagner G. Design Rationale of RuleML: A Markup Language for Semantic Web Rules, *Proc. Int. Semantic Web Working Symp.*, pp. 381-402, 2001.

7. Meditskos G, Bassiliades N. Towards an Object-Oriented Reasoning System for OWL, Int. Workshop on OWL Experiences and Directions, B. Cuenca Grau, I. Horrocks, B. Parsia, P. Patel-Schneider (Ed.), 11-12 Nov. 2005, Galway, Ireland, 2005.

8. Meditskos G, Bassiliades N. O-DEVICE: An Object-Oriented Knowledge Base System for OWL Ontologies, Proc. 4th Hellenic Conference on Artificial Intelligence (SETN-06), G. Antoniou, G. Potamias, D. Plexousakis, C. Spyropoulos (Ed.), Springer-Verlag, LNAI 3955, pp. 256-266, Heraklion, Crete, 18-20 May, 2006.

9. McBride B. Jena: Implementing the RDF Model and Syntax Specification, Proc. 2nd Int. Workshop on the Semantic Web, 2001.

10. OWL Web Ontology Language Reference, http://www.w3.org/TR/owl-ref/

11. Nute D. Defeasible Reasoning. *Proc. 20th Int. Conference on Systems Science* (pp. 470-477). IEEE Press. 1987.

12. Bassiliades N, Kontopoulos E, Antoniou G. A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web. *Proc. International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)*, A. Adi, S. Stoutenburg, S. Tabet (Ed.), Springer-Verlag, LNCS 3791, pp. 172-186, Galway, Ireland, 2005.

13. Bassiliades N, Antoniou G, Vlahavas I. A Defeasible Logic Reasoner for the Semantic Web. *Int. Journal on Semantic Web and Information Systems,* 2(1), pp. 1-41, 2006.

14. Kontopoulos E, Bassiliades N, Antoniou G, Visualizing Defeasible Logic Rules for the Semantic Web. *1st Asian Semantic Web Conference (ASWC'06) (to be presented)*, Beijing, China, 3-7 September, 2006.

15. Ullman J D. *Principles of Database and Knowledge-Base Systems*, Vol 1, Computer Science Press, 1988.

16. Sesame metadata repository, http://www.openrdf.org/

17. The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web, http://www.ics.forth.gr/isl/r-d-activities/rdf-gr.html