A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web

Nick Bassiliades[1], Efstratios Kontopoulos[1], Grigoris Antoniou[2]

[1]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
{nbassili, skontopo}@csd.auth.gr
[2]Institute of Computer Science, FO.R.T.H.
P.O. Box 1385, GR-71110, Heraklion, Greece
antoniou@ics.forth.gr

Abstract. Defeasible reasoning is a rule-based approach for efficient reasoning with incomplete and inconsistent information. Such reasoning is useful for many applications in the Semantic Web, such as policies and business rules, agent brokering and negotiation, ontology and knowledge merging, etc., mainly due to interesting features, such as conflicting rules and priorities of rules. However, the RuleML syntax of defeasible logic may appear too complex for many users. Furthermore, the interplay between various technologies and languages, such as defeasible reasoning, RuleML, and RDF impose a demand for using multiple, diverse tools for building rule-based applications for the Semantic Web. In this paper we present VDR-Device, a visual integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies. VDR-Device integrates in a user-friendly graphical shell, a visual RuleML-compliant rule editor that constrains the allowed vocabulary through analysis of the input RDF ontologies and a defeasible reasoning system that processes RDF data and RDF Schema ontologies.

1. Introduction

Although the Semantic Web represents a recent initiative to improve the potential of the current Web, it undoubtedly constitutes the inspiration for a vast number of applications. However, only the basic layers of the Semantic Web [9] have achieved a certain level of maturity, with the highest one of them being the ontology layer, where OWL [11], a description logic-based language, has become the dominant standard. The next layers that have to become more “concrete” are the logic and proof layers. Rule-based systems seem to possess a key role in this affair, since (a) they can serve as extensions of, or alternatives to, description logic based ontology languages; and (b) they can be used to develop declarative systems on top of (using) ontologies.

Defeasible reasoning [19], a member of the non-monotonic reasoning family, constitutes a simple rule-based approach to reasoning with incomplete and conflicting information. This approach offers two main advantages: (a) enhanced representational capabilities, allowing one to reason with incomplete and contradictory information,

coupled with (b) low computational complexity compared to mainstream non-monotonic reasoning. Defeasible reasoning can represent facts, rules as well as priorities and conflicts among rules. Such conflicts arise, among others, from rules with exceptions, which are a natural representation for policies and business rules [2]. And priority information is often implicitly or explicitly available to resolve conflicts among rules. Potential applications include security policies ([6], [16]), business rules [1], personalization, brokering [5], bargaining and agent negotiations ([13], [20]).

Although defeasible logic is certainly a very promising reasoning technology for the Semantic Web, the development of rule-based applications for the Semantic Web can be greatly compromised by two factors. First, defeasible logic is certainly not an end-user language but rather a developer's one, because its syntax (especially its RuleML compliant one presented in this paper) may appear too complex. Furthermore, the interplay between various technologies and languages involved in such applications, namely defeasible reasoning, RuleML, and RDF, impose a demand for using multiple, diverse tools, which is a high burden even for the developer.

In this paper we present VDR-Device, a visual integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies. VDR-Device integrates in a user-friendly graphical shell, a visual RuleML-compliant rule editor and a defeasible reasoning system that processes RDF data and RDF Schema ontologies [7]. The rule editor helps users to develop a defeasible logic rule base by constraining the allowed vocabulary after analyzing the input RDF ontologies. Therefore, it removes from the user the burden of typing-in class and property names and prevents potential semantical and syntactical errors. The visualization of rules follows the tree model of RuleML.

VDR-DEVICE supports multiple rule types of defeasible logic, as well as priorities among rules. Furthermore, it supports two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals. DR-DEVICE has a RuleML-compatible [10] syntax, which is the main standardization effort for rules on the Semantic Web. Input and output of data and conclusions is performed through processing of RDF data and RDF Schema ontologies. The system is built on-top of a CLIPS-based implementation of deductive rules, namely the R-DEVICE system [8]. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes.

The rest of the paper is organized as follows: Section 2 introduces a brokering trade case study that is used throughout the paper. Section 3 briefly introduces the semantics of defeasible logics. Section 4 presents the architecture and functionality of the VDR-Device system, including the visual rule editor and the core reasoning system. Finally, section 5 discusses related work and section 6 concludes the paper and discusses future work.

2. A Case Study

This section briefly presents a case study, adopted from [4], that is used throughout this paper to explicate the workings of defeasible logic and VDR-Device. The case study deals with a brokered trade application that takes place via an independent third

party, the broker, and more specifically with apartment renting. A number of available apartments reside in an RDF document along with the properties of each apartment (Fig. 1). The potential user expresses his/her requirements in defeasible logic (as explained in the following section), regarding the apartment he/she wishes to rent. The broker then tries to match the customer's requirements and the apartment specifications and proposes a deal when both parties can be satisfied by the trade.

```
<rdf:RDF ... xmlns:carlo="&carlo;" xmlns:carlo_ex="&carlo_ex;">
  <carlo:apartment rdf:about="&carlo_ex;a1">
    <carlo:bedrooms rdf:datatype="&xsd;integer">1</carlo:bedrooms>
    <carlo:central>yes</carlo:central>
    <carlo:floor rdf:datatype="&xsd;integer">1</carlo:floor>
    <carlo:gardenSize rdf:datatype="&xsd;integer">0</carlo:gardenSize>
    <carlo:lift>no</carlo:lift>
    <carlo:name>a1</carlo:name>
    <carlo:pets>yes</carlo:pets>
    <carlo:price rdf:datatype="&xsd;integer">300</carlo:price>
    <carlo:size rdf:datatype="&xsd;integer">50</carlo:size>
  </carlo:apartment>
  ...
</rdf:RDF>
```

Fig. 1. RDF document excerpt for available apartments.

The potential renter is looking for an apartment of at least $45m^2$ with at least 2 bedrooms. If it is on the 3rd floor or higher, the house must have an elevator. Also, pet animals must be allowed. He is willing to pay \$300 for a centrally located $45m^2$ apartment, and \$250 for a similar flat in the suburbs. In addition, he is willing to pay an extra \$5 per m^2 for a larger apartment, and \$2 per m^2 for a garden. He is unable to pay more than \$400 in total. If given the choice, he would go for the cheapest option. His 2nd priority is the presence of a garden; lowest priority is additional space.

3. Defeasible Logics - An Introduction

A *defeasible theory D* (i.e. a knowledge base or a program in defeasible logic) 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. *Strict rules* are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, so is the conclusion. An example of a strict rule is: "*Apartments are houses*", which, written formally, would become:

```
r1: apartment(X) → house(X)
```

Defeasible rules are rules that can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. An example of such a rule is "*Any apartment is considered to be acceptable*", which becomes `r2: apartment(X) ⇒ acceptable(X)`.

Defeaters, denoted by *A* ~> *p*, are rules that do not actively support conclusions, but can only prevent some of them. In other words, they are used to defeat some defeasible rules by producing evidence to the contrary. A defeater example is:

```
r3: ¬pets(X),gardenSize(x,y),y>0 ~> acceptable(X)
```

which reads as: "*If pets are not allowed in the apartment, but the apartment has a garden, then it might be acceptable*". This defeater can defeat, for example, rule `r4: ¬pets(X) ⇒ ¬acceptable(X)`.

Finally, a *superiority relationship* among the rule set R is an acyclic relation > on R. For example, given the defeasible rules r_2 and r_4, no conclusive decision can be made about whether the apartment is acceptable or not, because rules r_2 and r_4 contradict each other. But if a superiority relation > with $r_4 > r_2$ is introduced, then r_4 overrides r_2 and we can indeed conclude that the apartment is considered unacceptable. In this case rule r_4 is called *superior* to r_2 and r_2 *inferior* to r_4.

Another important element of defeasible reasoning is *conflicting literals*. In applications, literals are often considered to be conflicting and at most one of a certain set should be derived. An example of such an application is price negotiation, where an offer should be made by the potential buyer. The offer can be determined by several rules, whose conditions may or may not be mutually exclusive. All rules have `offer(X)` in their head, since an offer is usually a positive literal. However, only one offer should be made; therefore, only one of the rules should prevail, based on superiority relations among them. In this case, the conflict set is determined as follows:

```
C(offer(x,y)) = { ¬offer(x,y) } ∪ { offer(x,z) | z ≠ y }
```

For example, the following two rules make an offer for an given apartment, based on the buyer's requirements. However, the second one is more specific and its conclusion overrides the conclusion of the first one.

```
r5: size(X,Y),Y≥45,garden(X,Z) ⇒ offer(X,250+2+5(Y-45))
r6: size(X,Y),Y≥45,garden(X,Z),central(X) ⇒ offer(X,300+2Z+5(Y-45))
r6 > r5
```

4. VDR-Device System Architecture

The VDR-Device system consists of two primary components:

1. the reasoning system, named DR-Device, which performs the processing and inference and produces the results, and
2. the rule editor, named DRREd (Defeasible Reasoning Rule Editor), which serves both as a rule authoring tool and as a graphical shell for the core reasoning system.

Although these two subsystems utilize different technologies and were developed independently, they intercommunicate efficiently, forming a flexible and powerful integrated environment. The following subsections describe in detail these two major parts of the system.

4.1. Architecture and Functionality of the Reasoning System

The core reasoning system of VDR-Device is DR-Device [6] and consists of two primary components (Fig. 2): The *RDF loader/translator* and the *rule loader/translator*. The user can either develop a rule base (program, written in the RuleML-like syntax of VDR-Device) with the help of the rule editor described in the following sections,

or he/she can load an already existing one. 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.

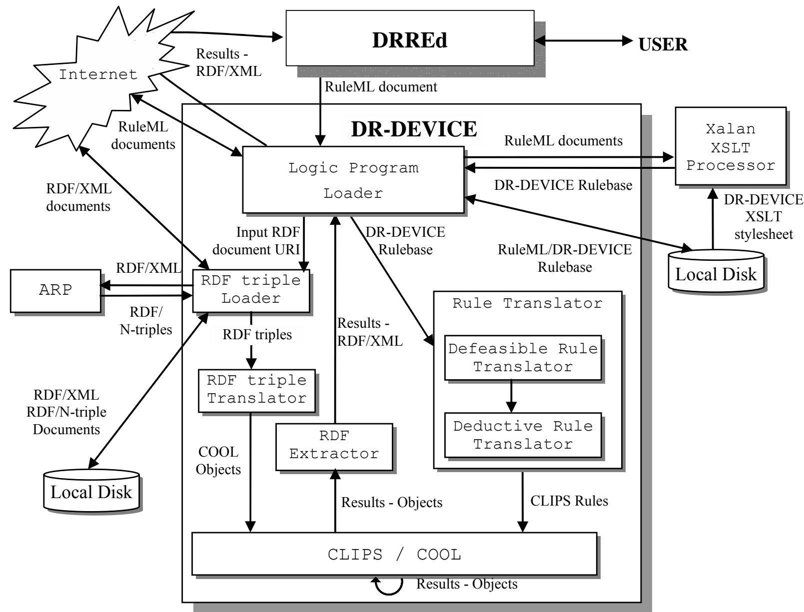

Fig. 2. The architecture of the core reasoning system.

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 [7]. Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in [8]. All compiled rule formats are also kept into local files (structured in project workspaces), so that the next time they are needed they can be directly loaded, improving speed considerably (running a compiled project is up to 10 times faster).

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 [8], which is briefly described below.

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 proven.

The Object-Oriented RDF Data Model

The DR-Device system employs an OO RDF data model, which is different than the established triple-based data model for RDF. The main difference is that DR-Device treats properties both as first-class objects and as normal encapsulated attributes of resource objects. In this way properties of resources are not scattered across several triples as in most other RDF inferencing systems, resulting in increased query performance due to less joins. For example, the apartment in Fig. 1 is transformed into the COOL object displayed in Fig. 3.

```
[carlo_ex:a1] of carlo:apartment
(carlo:size 50)                     (carlo:gardenSize 0)
(carlo:price 300)                   (carlo:floor 1)
(carlo:pets "yes")                  (carlo:central "yes")
(carlo:name "a1")                   (carlo:bedrooms 1)
(carlo:lift "no")
```

Fig. 3. COOL object for the apartment of Fig. 1.

The Defeasible Logic Language

DR-Device supports two syntaxes for defeasible logic rules: a native CLIPS-like one and a RuleML-compatible one. Here we focus solely on the latter, since the rule editor of the system allows the expression of rules only in this syntax. While the RuleML syntax utilizes as many features of the official RuleML as possible, several of the features of the rule language cannot be expressed by the existing RuleML DTDs and/or XML Schema documents. A new DTD (v. 0.85 compatible) and new XML Schema documents (0.86, 0.89 compatible) were, therefore, developed using the modularization scheme of RuleML, extending the Datalog with strong negation and negation-as-failure version of RuleML. Fig. 4 shows a self-contained simplified version of the DTD, while the original DTD and the XML Schema documents can be found at `http://lpis.csd.auth.gr/systems/dr-device.html`, along with the system itself. Notice, that currently the system uses the v. 0.85 compatible DTD.

A defeasible logic rule is represented by an `imp` element and consists of three subelements: the head and body of the rule (`_head` and `_body` elements respectively) as well as a label, encoded in a `_rlab` element, which includes the rule's unique ID (`ruleID` attribute) and its type (`ruletype` attribute). The latter can only take three distinct values (`strictrule`, `defeasiblerule`, `defeater`).

For example, the defeasible rule r_2 of the previous section is represented as:

```
<imp>
  <_rlab ruleID="r2" ruletype="defeasiblerule"><ind>r2</ind></_rlab>
  <_head>   <atom> <_opr><rel>acceptable</rel></_opr>
                   <_slot name="name"><var>X</var></_slot> </atom>
  </_head>
  <_body>   <atom> <_opr><rel href="carlo:apartment"/></_opr>
                   <_slot name="name"><var>X</var></_slot> </atom>
  </_body>
```

```
</imp>
```

The names (`rel` elements) of the operator (`_opr`) elements of atoms are class names, since atoms actually represent CLIPS objects. RDF class names used as base classes in the rule condition are referred to through the `href` attribute of the `rel` element, while derived class names (e.g. `acceptable`) are text values of the `rel` element. Atoms have named arguments, called slots, which correspond to object properties. Since RDF resources are represented as CLIPS objects, atoms correspond to queries over RDF resources of a certain class with certain property values.

```
<!ENTITY % URI "CDATA">
<!ELEMENT rulebase (_rbaselab, (imp | comp_rules)*)>
<!ATTLIST rulebase rdf_import CDATA #IMPLIED
                   rdf_export_classes NMTOKENS #IMPLIED
                   rdf_export CDATA #IMPLIED>
<!ELEMENT _rbaselab (ind)>
<!ELEMENT imp (_rlab, _head, _body)>
<!ELEMENT comp_rules (_crlab)>
<!ATTLIST comp_rules c_rules IDREFS #REQUIRED
                     slotnames NMTOKENS #IMPLIED>
<!ELEMENT _rlab (ind)>       <!ELEMENT _crlab (ind)>
<!ATTLIST _rlab  ruleID ID #REQUIRED
           ruletype (strictrule | defeasiblerule | defeater) #REQUIRED
           superior IDREFS #IMPLIED>
<!ELEMENT _head (calc?, (atom | neg))>
<!ELEMENT _body (atom | neg | and)>
<!ELEMENT calc (fun_call+)>
<!ELEMENT fun_call (ind|var|fun_call)*>
<!ATTLIST fun_call   name CDATA #REQUIRED>
<!ELEMENT naf (atom | and)>          <!ELEMENT neg (atom)>
<!ELEMENT and ((atom | naf)*)>       <!ELEMENT atom (_opr, _slot*)>
<!ELEMENT _opr (rel)>                <!ELEMENT rel (#PCDATA)>
<!ATTLIST rel    href %URI; #IMPLIED>
<!ELEMENT _slot (ind | var | _not | _or | _and)>
<!ATTLIST _slot  name CDATA #REQUIRED>
<!ELEMENT _not (ind | var)>
<!ELEMENT _or ((_not|ind|var|fun_call),(_not|ind|var|fun_call)+)>
<!ELEMENT _and ((_not|ind|var|fun_call),(_not|ind|var|fun_call)+)>
<!ELEMENT ind (#PCDATA)>             <!ELEMENT var (#PCDATA)>
<!ATTLIST ind    type CDATA #IMPLIED  href %URI; #IMPLIED>
```

Fig. 4. DTD for the RuleML syntax of the defeasible logic rule language.

Superiority relations are represented as attributes of the superior rule. For example, rule r_4, which is superior to r_2, is represented as follows:

```
<imp>
  <_rlab ruleID="r4" ruletype="defeasiblerule" superior="r2">
         <ind>r4</ind> </_rlab>
  <_head><neg> <atom> <_opr><rel>acceptable</rel></_opr>
                      <_slot name="name"><var>X</var></_slot> </atom>
         </neg> </_head>
  <_body>  <atom> <_opr><rel href="carlo:apartment"/></_opr>
                  <_slot name="carlo:name"><var>X</var></_slot>
                  <_slot name="carlo:pets"><ind>no</ind></_slot>
           </atom>  </_body>
</imp>
```

Negation is represented via a neg element that encloses an atom element. Apart from rule declarations, there are comp_rules elements that declare groups of competing rules which derive competing positive conclusions (*conflicting literals*). For example, in the apartment rent case study, rules r_5 and r_6 are competing over the offer(X,Y) conclusion, since at most one offer can be made:

```
<comp_rules c_rules="r5 r6">
  <_crlab> <ind>cr1</ind> </_crlab>
</comp_rules>
```

Further extensions to the RuleML syntax, include function calls that are used either as constraints in the rule body or as new value calculators at the rule head. Additionally, multiple constraints in the rule body can be expressed through the logical operators: _not, _and, _or. Finally, the header of the rule base, namely the rulebase root element of the RuleML document, includes a number of important parameters, which are implemented as attributes: rdf_import declares the input RDF file(s), rdf_export represents the RDF file that contains the exported results and rdf_export_classes represents the derived classes, whose instances will be exported in RDF/XML format. An example of all of the above is shown below:

```
<rulebase rdf_import="http://lpis.csd.auth.gr/.../carlo.rdf#"
          rdf_export="http://lpis.csd.auth.gr/.../export-carlo.rdf"
          rdf_export_classes="acceptable rent">
```

4.2. Rule Editor – Design and Functionality

Although RuleML syntax improves readability on behalf of human users, writing rules in RuleML can often be a highly cumbersome task. Thus, the need for authoring tools that assist end-users in writing and expressing rules is apparently imperative. VDR-Device is equipped with DRREd, a Java-built visual rule editor that aims at enhancing user-friendliness and efficiency during the development of VDR-Device RuleML documents. Its implementation is oriented towards simplicity of use and familiarity of interface. Other key features of the software 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 with only a few steps and (c) powerful safety mechanisms – the correct syntax is ensured and the user is protected from syntactical or semantical errors.

More specifically, and as can be observed in Fig. 5, the main window of the program is composed of two major parts: a) the upper part includes the menu bar, which contains the program menus, and the toolbar that includes icons, representing the most common utilities of the rule editor, and b) the central and more “bulky” part is the primary frame of the main window and is in turn divided into two panels:

The left panel displays the rule base in XML-tree format, which is the most intuitive means of displaying RuleML-like syntax, because of its hierarchical nature. The user has the option of navigating through the entire 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 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.

The right panel shows a table, which contains the attributes that correspond each time to the selected tree node in the left-hand area. The user can also perform editing functions on the attributes, by altering the value for each attribute in the panel that appears below the attributes table on the right-hand side. The values that the user can insert are obviously limited by the chosen attribute each time.

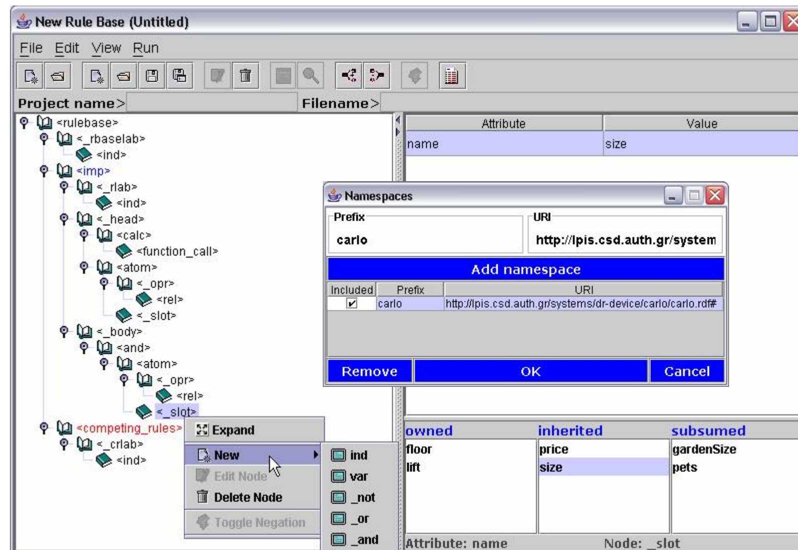

Fig. 5. The graphical rule editor and the namespace dialog window.

The development of a rule base using VDR-Device is a delicate process that depends heavily on the parameters around the node that is being edited each time. First of all, there is an underlying principle behind tree expansion and it is “triggered” each time the user is trying to add a new element to the rule base. Namely, when a new element is added to the tree, all the mandatory sub-elements that accompany it are also added. In the cases where there are multiple sub-elements, none of them is added to the rule base and the final choice is left to the user to determine which one of them has to be added. The user has to right-click on the parent element and choose the desired sub-element from the pop-up menu that appears (Fig. 5).

Another important aspect of the rule editor is the namespace dialog window (Fig. 5), where the user can determine which RDF/XML namespaces will be used by the rule base. Actually, we treat namespace declarations as addresses of input RDF Schema ontologies that contain the vocabulary for the input RDF documents, over which the rules of the rule base will be run. The namespaces entered by the user, as well as those contained in the input RDF documents (indicated by the `rdf_import` attribute of the `rulebase` root element), are then analyzed in order to extract all the allowed class and property names for the rule base being developed (see next section). These names are then used throughout the authoring phase of the RuleML rule base, constraining the corresponding allowed names that can be applied and narrowing the possibility for errors on behalf of the user.

Moving on to more node-specific features of the rule editor, one of the rule base elements that are treated in a specific manner is the `atom` element, which can be either negated or not. The response of the editor to an atom negation is performed through the wrapping/unwrapping of the `atom` element within a `neg` element and it is performed via a toggle button, located on the overhead toolbar.

Some components that also need "special treatment" are the rule IDs, each of which uniquely represents a rule within the rule base. Thus, the rule editor has to collect all of the RuleIDs inserted, in order to prohibit the user from entering the same RuleID twice and also equipping other IDREF attributes (e.g. `superior` attribute) with the list of RuleIDs, constraining the variety of possible values.

The names of the functions that appear inside a `fun_call` element are also partially constrained by the rule editor, since the user can either insert a custom-named function or a CLIPS built-in function. Through radio-buttons the user determines whether he/she is using a custom or a CLIPS function. In the latter case, a list of all built-in functions is displayed, once again constraining possible entries.

Finally, users can examine all the exported results via an Internet Explorer window, launched by VDR-Device. Also, to improve reliability, the user can also observe the execution trace of compilation and running, both during run-time and also after the whole process has been terminated (Fig. 6).

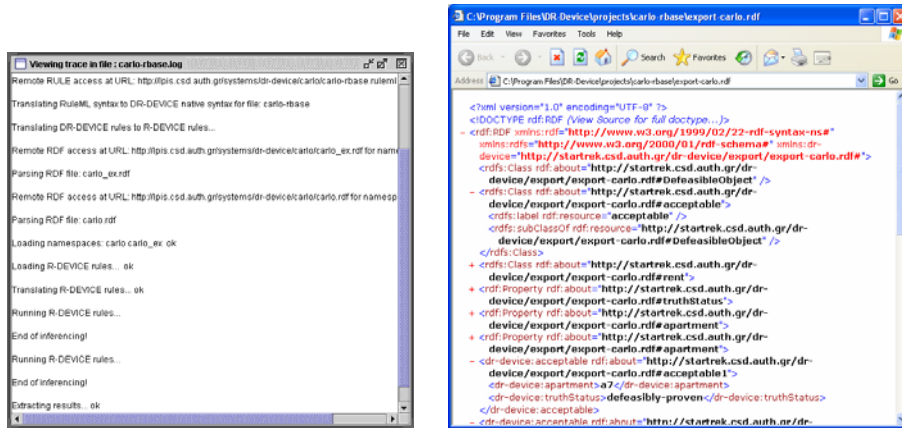

Fig. 6. The Trace and Results windows.

Parsing RDF Schema Ontologies

As mentioned above, the RDF Schema documents contained in the namespace dialog window undergo certain processing and, more specifically, they are being parsed, using the ARP parser of Jena [18], a flexible Java API for processing RDF documents. The names of the classes found are collected in the *base class vector* (CV_b), which already contains `rdfs:Resource`, the superclass of all RDF user classes. Therefore, the CV_b vector is constructed as follows:

`rdfs:Resource` $\in CV_b$

$\forall C\,(C$ `rdf:type rdfs:Class`$) \rightarrow C \in CV_b$

where ($X\ Y\ Z$) represents an RDF triple found in the RDF Schema documents.

Except from the base class vector, there also exists the *derived class vector* (CV_d), which contains the names of the derived classes, i.e. the classes which lie at rule heads (*conclusions*). CV_d is initially empty and is dynamically extended every time a new class name appears inside the `rel` element of the `atom` in a rule head. This vector is mainly used for loosely suggesting possible values for the `rel` elements in the rule head, but not constraining them, since rule heads can either introduce new derived classes or refer to already existing ones.

The union of the above two vectors results in CV_f, which is the *full class vector* ($CV_f = CV_b \cup CV_d$) and it is used for constraining the allowed class names, when editing the contents of the `rel` element inside `atom` elements of the rule body.

Furthermore, the RDF Schema documents are also being parsed for property names and their domains. Similarly to the procedure described above, the properties detected are placed in a *base property vector* (PV_b), which already contains some built-in RDF properties (*BIP*) whose domain is `rdfs:Resource`:

BIP = {`rdf:type`, `rdfs:label`, `rdfs:comment`, `rdfs:seeAlso`,
`rdfs:isDefinedBy`, `rdf:value`} $\subseteq PV_b$

$\forall P$, (P `rdf:type rdf:Property`) $\rightarrow P \in PV_b$

Apparently, there also exists the *derived property vector* (PV_d), which contains the names of the properties of the derived classes. This vector is initially empty and is extended each time a new property name appears inside the `_slot` element of the `atom` in a rule head. Therefore, the *full property vector* (PV_f) is a union of the above two vectors: $PV_f = PV_b \cup PV_d$.

Each of the properties in the PV_f vector has to be equipped with the corresponding superproperties and domains. Through the detected superproperties, the system can retrieve the indirect domains for each property and, thus, enrich its set of domains. The domain set of each property is needed, so that, for each `atom` element appearing inside the rule body, when a specific class C is selected, the names of the properties that can appear inside the `_slot` subelements are constrained only to those that have C as their domain, either directly or inherited.

So, the *superproperty set* $SUPP(P)$ of each property P initially contains only the direct superproperties of P. The rest of the properties (including the derived class properties) have an empty $SUPP(P)$:

$\forall P \in PV_b\ \forall SP \in PV_b$, ($P$ `rdfs:subPropertyOf` SP) $\rightarrow SP \in SUPP(P)$

In the next step, the $SUPP(P)$ set is further populated with the indirect superproperties of each property, by recursively traversing upwards the property hierarchy:

$\forall P \in PV_b\ \forall SP \in SUPP(P)\ \forall SP' \in SUPP(SP) \rightarrow SP' \in SUPP(P)$

On the other hand, the $DOM(P)$ set of domains for each property P initially contains only the direct domain of P:

$\forall P \in PV_b\ \forall C$, ($P$ `rdfs:domain` C) $\rightarrow C \in DOM(P)$

The RDF built-in properties (*BIP*) have `rdfs:Resource` as their domain:

$\forall P \in BIP$, `rdfs:Resource` $\in DOM(P)$

If a property does not have a domain, then `rdfs:Resource` is assumed:

$\forall\ P \in (PV_b\text{-}BIP), (\neg\exists C\ P\ \texttt{rdfs:domain}\ C) \rightarrow \texttt{rdfs:Resource} \in DOM(P)$

In the next step, the $DOM(P)$ set is further populated, by inheriting the domains of all the superproperties (both direct and indirect), according to the RDFS semantics:

$\forall P \in PV_b\ \forall SP \in SUPP(P)\ \forall C \in DOM(SP) \rightarrow C \in DOM(P)$

Since the properties are now fully described (each one of them containing the corresponding superproperty and domain sets), each class C in the CV_f vector has to be linked with the allowed properties. More specifically, for each class C, five distinct sets have to be defined: *superclass set* $SUPC(C)$, *subclass set* $SUBC(C)$, *owned property set* $OWNP(C)$, *inherited property set* $INHP(C)$, and *subsumed property set* $SUBP(C)$.

The $SUPC(C)$ set initially contains all the direct superclasses of C:

$\forall C \in CV_f\ \forall SC \in CV_f, (C\ \texttt{rdfs:subClassOf}\ SC) \rightarrow SC \in SUPC(C)$

If a class does not have a superclass, then it is considered to be a subclass of `rdfs:Resource`. This also applies for the derived classes:

$$\forall C \in CV_f, C \neq \texttt{rdfs:Resource} \wedge (\neg\exists SC\ SC \in CV_f \rightarrow (C\ \texttt{rdfs:subClassOf}\ SC)) \\ \rightarrow \texttt{rdfs:Resource} \in SUPC(C)$$

In the next phase, the $SUPC(C)$ set is further populated with the indirect superclasses of each class, by recursively traversing upwards the class hierarchy:

$\forall C \in CV_f\ \forall SC \in SUPC(C)\ \forall SC' \in SUPC(SC) \rightarrow SC' \in SUPC(C)$

The $SUBC(C)$ set can now be easily constructed, by inversing all the subclass relationships (both direct and indirect):

$\forall C \in CV_f\ \forall SC \in SUPC(C) \rightarrow C \in SUBC(SC)$

The $OWNP(C)$ set of owned properties is constructed, by examining the domain set of each property object in the full property vector:

$\forall P \in PV_f\ \forall C \in DOM(P) \rightarrow P \in OWNP(C)$

The inherited property set $INHP(C)$ is constructed, by inheriting the owned properties from all the superclasses (both direct and indirect), according again to the RDFS semantics:

$\forall C \in CV_b\ \forall SC \in SUPC(C)\ \forall\ P \in OWNP(SC) \rightarrow P \in INHP(C)$

Finally, the subsumed property set $SUBP(C)$ is constructed, by copying the owned properties from all the subclasses (both direct and indirect):

$\forall C \in CV_b\ \forall SC \in SUBC(C)\ \forall P \in OWNP(SC) \rightarrow P \in SUBP(C)$

Although the domain of a subsumed property of a class C is not compatible with class C, it can still be used in the rule condition for querying objects of class C, implying that the matched objects will belong to some subclass C' of class C, which is compatible with the domain of the subsumed property. For example, consider two classes A and B, the latter being a subclass of the former, and a property P, whose domain is B. It is allowed to query class A, demanding that property P satisfies a certain condition; however, only objects of class B can possibly satisfy the condition, since direct instances of class A do not even have property P.

The above mentioned three property sets comprise the *full property set* $FPS(C)$:

$FPS(C) = OWNP(C) \cup INHP(C) \cup SUBP(C)$

which is used to restrict the names of properties that can appear inside a `_slot` element (see Fig. 5), when the class of the atom element is *C*.

An example of all of the above is shown in Table 1. Assume an RDF Schema ontology with three classes connected through a hierarchy: the class `apartment` is a subclass of the `house` class and a superclass of the `suburban-apartment` class. Some typical properties of these classes are displayed in the row "owned properties" of Table 1.

After the RDF Schema document is parsed, these classes are detected and included in the *base class vector* (CV_b). Furthermore, the corresponding properties are determined and added to the *base property vector* (PV_b). Eventually, every available class will be linked to the respective properties, but also to the properties of its super- and subclasses, following the rationale developed before in this section. The final status of the class properties is displayed in Table 1.

Table 1. Example of inherited, owned and subsumed properties.

Classes	**house**	**apartment**	**suburban-apartment**
Owned Properties	size price	floor lift	gardenSize pets
Inherited Properties		size price	size, price floor, lift
Subsumed Properties	floor, lift gardenSize, pets	gardenSize pets	

This logic is reflected in the rule editor, as Fig. 5 shows. If, for example, the user wishes to formulate the rule r_4 (section 4.1), then he/she selects the `carlo:apartment` class as the value of the `href` attribute of the `_opr` element of an atom in the rule body and the allowed properties to be entered at the `_slot` element are all the properties included in Table 1. This facilitates the user, since he/she does not have to worry about which properties can be applied to apartment instances.

5. Related Work

There exist several previous implementations of defeasible logics, although to the best of our knowledge none of them is supported by a user-friendly integrated development environment or a visual rule editor. *Deimos* [17] is a flexible, query processing system based on Haskell. It implements several variants, but not conflicting literals nor negation as failure in the object language. Also, it does not integrate with Semantic Web (for example, there is no way to treat RDF data and RDFS/OWL ontologies; nor does it use an XML-based or RDF-based syntax for syntactic interoperability). Therefore, it is only an isolated solution. Finally, it is propositional and does not support variables.

Delores [17] is another implementation, which computes all conclusions from a defeasible theory. It is very efficient, exhibiting linear computational complexity. Delores only supports ambiguity blocking propositional defeasible logic; so, it does not support ambiguity propagation, nor conflicting literals, variables and negation as

failure in the object language. Also, it does not integrate with other Semantic Web languages and systems, and is thus an isolated solution.

SweetJess [15] is another implementation of a defeasible reasoning system (situated courteous logic programs) based on Jess. It integrates well with RuleML. However, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML-OIL like syntax of RuleML) and not on arbitrary RDF data, like DR-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms). This applies to DR-DEVICE to a large extent. However, the basic R-DEVICE language [8] can support a limited form of functions in the following sense: (a) path expressions are allowed in the rule condition, which can be seen as complex functions, where allowed function names are object referencing slots; (b) aggregate and sorting functions are allowed in the conclusion of aggregate rules. Finally, DR-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules [7].

Mandarax [12] is a Java rule platform, which provides a rule mark-up language (compatible with RuleML) for expressing rules and facts that may refer to Java objects. It is based on derivation rules with negation-as-failure, top-down rule evaluation, and generating answers by logical term unification. RDF documents can be loaded into Mandarax as triplets. Furthermore, Mandarax is supported by the Oryx graphical rule management tool. Oryx includes a repository for managing the vocabulary, a formal-natural-language-based rule editor and a graphical user interface library. Contrasted, the rule authoring tool of DR-DEVICE lies closer to the XML nature of its rule syntax and follows a more traditional object-oriented view of the RDF data model [8]. Furthermore, DR-DEVICE supports both negation-as-failure and strong negation, and supports both deductive and defeasible logic rules.

6. Conclusions and Future Work

In this paper we argued that defeasible reasoning is useful for many applications in the Semantic Web, mainly due to conflicting rules and rule priorities. However, the development of defeasible rule bases on top of Semantic Web ontologies may appear too complex for many users. To this end, we have implemented VDR-Device, a visual integrated development environment that facilitates these processes. VDR-Device features a user-friendly graphical shell, a visual RuleML-compliant rule editor that helps users to develop a defeasible logic rule base by constraining the allowed vocabulary after analyzing the input RDF ontologies and a defeasible reasoning system that supports direct import from the Web and processing of RDF data and RDF Schema ontologies.

In the future, we plan to delve into the proof layer of the Semantic Web architecture by enhancing further the graphical environment with rule execution tracing, explanation, proof exchange in an XML or RDF format, proof visualization and validation, etc. These facilities would be useful for increasing the trust of users for the Semantic Web agents and for automating proof exchange and trust among agents in the Semantic Web. Furthermore, we will include a graphical RDF ontology and data editor that will comply with the user-interface of the RuleML editor.

Acknowledgements

This work is partially supported by the European IST Network of Excellence REWERSE and by the PYTHAGORAS II programme which is jointly funded by the Greek Ministry of Education (EPEAEK) and the European Union.

References

[1] Antoniou G. and Arief M., "Executable Declarative Business rules and their use in Electronic Commerce", *Proc. ACM Symposium on Applied Computing*, 2002.
[2] Antoniou G., Billington D. and Maher M.J., "On the analysis of regulations using defeasible rules", *Proc. 32nd Hawaii International Conference on Systems Science*, 1999.
[3] Antoniou G., Billington D., Governatori G. and Maher M.J., "Representation results for defeasible logic", *ACM Trans. on Computational Logic*, 2(2), 2001, pp. 255-287.
[4] Antoniou G., Harmelen F. van, *A Semantic Web Primer*, MIT Press, 2004.
[5] Antoniou G., Skylogiannis T., Bikakis A., Bassiliades N., "DR-BROKERING – A Defeasible Logic-Based System for Semantic Brokering", *IEEE Int. Conf. on E-Technology, E-Commerce and E-Service*, pp. 414-417, Hong Kong, 2005.
[6] Ashri R., Payne T., Marvin D., Surridge M. and Taylor S., "Towards a Semantic Web Security Infrastructure", *Proc. of Semantic Web Services*, 2004 Spring Symposium Series, Stanford University, California, 2004.
[7] Bassiliades N., Antoniou, G., Vlahavas I., "A Defeasible Logic Reasoner for the Semantic Web", *RuleML 2004*, Springer-Verlag, LNCS 3323, pp. 49-64, Hiroshima, Japan, 2004.
[8] Bassiliades N., Vlahavas I., "R-DEVICE: A Deductive RDF Rule Language", *RuleML 2004*, Springer-Verlag, LNCS 3323, pp. 65-80, Hiroshima, Japan, 2004.
[9] Berners-Lee T., Hendler J., and Lassila O., "The Semantic Web", *Scientific American*, 284(5), 2001, pp. 34-43.
[10] Boley H., Tabet S., *The Rule Markup Initiative*, `www.ruleml.org/`
[11] Dean M. and Schreiber G., (Eds.), OWL Web Ontology Language Reference, 2004, `www.w3.org/TR/2004/REC-owl-ref-20040210/`
[12] Dietrich J., Kozlenkov A., Schroeder M., Wagner G., "Rule-based agents for the semantic web", *Electronic Commerce Research and Applications*, 2(4), pp. 323–338, 2003.
[13] Governatori G., Dumas M., Hofstede A. ter and Oaks P., "A formal approach to legal negotiation", *Proc. ICAIL 2001*, pp. 168-177, 2001.
[14] Grosof B. N., "Prioritized conflict handing for logic programs", *Proc. of the 1997 Int. Symposium on Logic Programming*, pp. 197-211, 1997.
[15] Grosof B.N., Gandhe M.D., Finin T.W., "SweetJess: Translating DAMLRuleML to JESS", *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML 2002)*.
[16] Li N., Grosof B. N. and Feigenbaum J., "Delegation Logic: A Logic-based Approach to Distributed Authorization", *ACM Trans. on Information Systems Security*, 6(1), 2003.
[17] Maher M.J., Rock A., Antoniou G., Billington D., Miller T., "Efficient Defeasible Reasoning Systems", *Int. Journal of Tools with Artificial Intelligence*, 10(4), 2001, pp. 483-501.
[18] McBride B., "Jena: Implementing the RDF Model and Syntax Specification", *Proc. 2nd Int. Workshop on the Semantic Web*, 2001.
[19] Nute D., "Defeasible Reasoning", *Proc. 20th Int. Conference on Systems Science*, IEEE Press, 1987, pp. 470-477.
[20] Skylogiannis T., Antoniou G., Bassiliades N., Governatori G., "DR-NEGOTIATE – A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies", *IEEE Int. Conf. on E-Technology, E-Commerce and E-Service*, pp. 44-49, Hong Kong, 2005.