3.5 A Non-Monotonic Reasoning System for RDF Metadata

A Non-Monotonic Reasoning System for RDF Metadata

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

[1]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
{ skontopo,nbassili}@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

Non-monotonic reasoning constitutes an approach to reasoning with incomplete or changing information and is significantly more powerful than standard reasoning, which simply deals with universal statements. Defeasible reasoning, a member of the non-monotonic reasoning family, offers the extra capability of dealing with conflicting information and can represent facts, rules and priorities among rules. The main advantages of defeasible reasoning, however, are not only limited to its enhanced representational capabilities, but also feature low computational complexity compared to mainstream non-monotonic reasoning. This paper presents a system for non-monotonic reasoning on the Semantic Web called VDR-Device, which is capable of reasoning about RDF metadata over multiple Web sources using defeasible logic rules. It is implemented on top of the CLIPS production rule system and features a RuleML compatible syntax. The operational semantics of defeasible logic are implemented through compilation into a generic deductive rule language. Since the RuleML syntax may appear complex for many users, we have also implemented a graphical authoring tool for defeasible logic rules that acts as a shell for the defeasible reasoning system. The tool constrains the allowed vocabulary through analysis of the input RDF documents, so that the user does not have to type-in class and property names.

Introduction

The development of the Semantic Web (Berners-Lee, Hendler and Lassila 2001) proceeds in a hierarchy of layers, with each layer being on top of other layers. At present, the highest layer that has reached sufficient maturity is the ontology layer, where OWL (Dean and Schreiber 2004), a description logic based language, is currently the dominant standard.

Above the ontology layer lie the logic and proof layers, towards which the next steps in the development of the Semantic Web will be directed. Rule systems can play a twofold role in the Semantic Web initiative: (a) they can serve as extensions of, or alternatives to, description logic based ontology languages, since rules are more expressive than description logic languages like OWL and (b) they can be used to develop declarative systems on top of (using) ontologies.

Non-monotonic reasoning (Antoniou 1997) constitutes an approach that allows reasoning with incomplete or changing information. More specifically, it provides mechanisms for taking back conclusions that, in the presence of new information, turn out to be wrong and for deriving new, alternative conclusions instead. Contrary to standard reasoning, which simply deals with universal statements, non-monotonic reasoning offers a significantly higher level of expressiveness.

Defeasible reasoning (Nute 1987), a member of the non-monotonic reasoning family, represents a simple rule-based approach to reasoning not only with incomplete or changing but also with conflicting information. When compared to mainstream non-monotonic reasoning, the main advantages of defeasible reasoning are enhanced representational capabilities coupled with low computational complexity.

Defeasible reasoning can represent facts, rules and priorities and conflicts among rules. Such conflicts arise, among others, from rules with exceptions, which are a natural representation for policies and business rules (Antoniou, Billington and Maher 1999) and priority information is often available to resolve conflicts among rules. Other application domains are described later on in this work.

In this paper we report on the implementation of VDR-DEVICE which is a visual, integrated environment for the development and application of defeasible logic rule bases on top of RDF ontologies. VDR-Device consists of: (i) a visual RuleML-compliant rule editor and (ii) a defeasible reasoning system for the Semantic Web that processes RDF data and RDF Schema ontologies.

VDR-Device supports multiple rule types of defeasible logic (strict rules, defeasible rules and defeaters) as well as priorities among rules. It also supports two types of negation (strong negation and negation-as-failure) and conflicting (mutually exclusive) literals.

The system has a RuleML-compatible (Boley et al. 2001) syntax, which expresses the main standardization effort for rules in the Semantic Web. Input and output of data is performed through processing of RDF data and RDF Schema ontologies.

The reasoning system of VDR-Device is built on-top of a CLIPS-based implementation of deductive rules, called R-Device (Bassiliades and Vlahavas 2006). The core mechanism of the system performs the translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes.

The rest of the paper is organized as follows: Firstly, the motivation for utilizing defeasible reasoning in the Semantic Web is more thoroughly examined. Then, a brief introduction to defeasible logics is made, followed by the section that presents the VDR-Device system. The presentation includes the architecture and functionality of the system, the syntax of the defeasible logic rule language, the underlying core deductive rule language, the translation from the defeasible logic rules to the deductive rules and the graphical rule editor. Related work on defeasible reasoning systems is discussed, next. Finally, this paper sums up the conclusions and gives potential directions for future work.

Conflicting Rules in the Semantic Web

This section briefly describes the main cases, where conflicting rules might be applied in the Semantic Web.

Reasoning with Incomplete Information

In (Antoniou 2002) a scenario is described where business rules have to deal with incomplete information: in the absence of certain information some assumptions have to be made that lead to conclusions not supported by classical predicate logic. In many applications on the Web such assumptions must be made because other players may not be able (e.g. due to communication problems) or willing (e.g. because of privacy or security concerns) to provide information. This is the classical case for the use of non-monotonic knowledge representation and reasoning (Marek and Truszczynski 1993).

Rules with Exceptions

As mentioned earlier, rules with exceptions are a natural way of representation for policies and business rules. And priority information is often implicitly or explicitly available to resolve conflicts among rules. Potential applications include security policies (Ashri et al. 2004), business rules (Antoniou and Arief 2002), e-contracting (Governatori 2005), brokering (Antoniou et al. 2005) and agent negotiations (Governatori et al. 2001).

Default Inheritance in Ontologies

Default inheritance is a well-known feature of certain knowledge representation formalisms. Thus it may play a role in ontology languages, which currently do not support this feature. In (Grosof and Poon 2003) some ideas are presented for possible uses of default inheritance in ontologies. A natural way of representing default inheritance is rules with exceptions plus priority information. Thus, non-monotonic rule systems can be utilized in ontology languages.

Ontology Merging

When ontologies from different authors and/or sources are merged, contradictions arise naturally. Predicate logic based formalisms, including all current Semantic Web languages, cannot cope with inconsistencies. If rule-based ontology languages are used (e.g. DLP (Grosof et al. 2003)) and if rules are interpreted as defeasible (that is, they may be prevented from being applied even if they can fire) then we arrive at non-monotonic rule systems. A skeptical approach, as adopted by defeasible reasoning, is sensible because it does not allow for contradictory conclusions to be drawn. Moreover, priorities may be used to resolve some conflicts among rules, based on knowledge about the reliability of sources or on user input). Thus, non-monotonic rule systems can support ontology integration.

Defeasible Logics

A *defeasible theory D* is a couple (*R*,>) where *R* a finite set of rules, and > a superiority relation on R. In expressing the proof theory we consider only propositional rules. Rules containing free variables are interpreted as the set of their variable-free instances.

There are three kinds of rules: *Strict rules* are denoted by $A \rightarrow p$, and are interpreted in the classical sense: whenever the premises are indisputable then so is the conclusion. An example of a strict rule is “Professors are faculty members”. Written formally:

```
professor(X) → faculty(X).
```

Inference from strict rules only is called *definite inference*. Strict rules are intended to define relationships that are definitional in nature, for example ontological knowledge.

Defeasible rules are denoted by $A \Rightarrow p$ and can be defeated by contrary evidence. An example of such a rule is:

```
professor(X) ⇒ tenured(X)
```

which reads as follows: “Professors are typically tenured”.

Defeaters are denoted as *A* ~> *p* and cannot actively support conclusions, but are used only to prevent some of them. A defeater example is:

```
assistantProf(X) ~> ¬tenured(X)
```

which is interpreted as follows: “Assistant professors may be not tenured”.

A *superiority relation* on R is an acyclic relation > on R (that is, the transitive closure of > is irreflexive). When $r_1 > r_2$, then r_1 is called *superior* to r_2, and r_2 *inferior* to r_1.

This expresses that r_1 may override r_2. For example, given the defeasible rules

```
r₁: professor(X) ⇒  tenured(X)
r₂: visiting(X)  ⇒ ¬tenured(X)
```

which contradict one another, no conclusive decision can be made about whether a visiting professor is tenured. But if we introduce a superiority relation > with $r_2 > r_1$, then we can indeed conclude that a visiting professor is not tenured.

Another important element of defeasible reasoning is the notion of *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:

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

For example, the following two rules make an offer for a 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+2Z+5(Y-45))
r₆: size(X,Y),Y≥45,garden(X,Z),central(X) ⇒
                                 offer(X,300+2Z+5(Y-45))
r₆ > r₅
```

The VDR-Device System

The VDR-Device system consists of two primary components:

1. DR-Device, the reasoning system that performs the RDF processing and inference and produces the results, and
2. DRREd (Defeasible Reasoning Rule Editor), the 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 Non-Monotonic Reasoning System

The core reasoning system of VDR-Device is DR-Device (Bassiliades, Antoniou and Vlahavas 2006) and consists of two primary components (Fig. 1): 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 a following section, 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.

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 (`http://www.ghg.net/clips/CLIPS.html`). 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 (Bassiliades, Antoniou and Vlahavas 2006). Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in (Bassiliades and Vlahavas 2006). All compiled rule formats are also kept in 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).

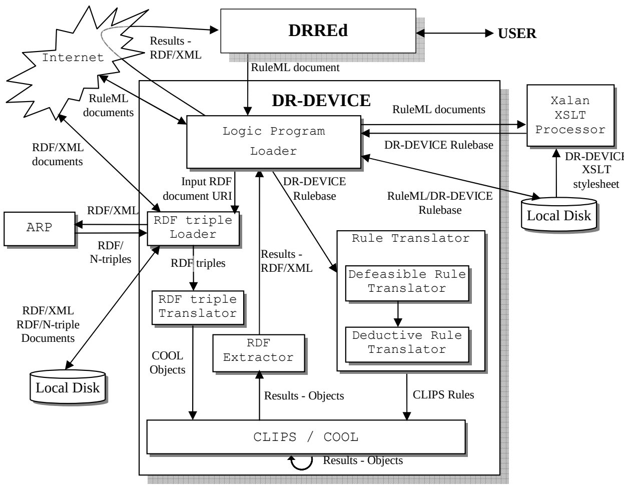


Fig. 1. Architecture of the VDR-DEVICE system.

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 (Bassiliades and Vlahavas 2006), 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 proved.

Syntax of the Defeasible Logic Rule Language

There are three types of rules in DR-DEVICE, closely reflecting defeasible logic: strict rules, defeasible rules, and defeaters. Each rule type is declared with a corresponding keyword (`strictrule`, `defeasiblerule` and `defeater` respectively). For example, the following rule construct represents the defeasible rule r_1: `professor(X)` ⇒ `tenured(X)`.

```
(defeasiblerule r1
   (professor (name ?X))
⇒
   (tenured (name ?X)))
```

Predicates have named arguments, called slots, since they represent CLIPS objects. DR-DEVICE has also a RuleML-like syntax. The same rule is represented in RuleML notation (version 0.85) as follows:

```
<imp>
   <_rlab ruleID="r1" ruletype="defeasiblerule">
      <ind>r1</ind></_rlab>
   <_head>
      <atom><_opr><rel>professor</rel></_opr>
            <_slot name="name"/><var>X</var></_slot>
      </atom></_head>
   <_body>
      <atom><_opr><rel href="tenured"/></_opr>
            <_slot name="name"><var>X</var></_slot>
      </atom>
   </_body>
</imp>
```

We have tried to re-use as many features of RuleML syntax as possible. However, several features of the DR-DEVICE rule language could not be captured by the existing RuleML DTDs (version 0.9); therefore, we have developed a new DTD (Fig. 2), using the modularization scheme of RuleML, extending the Datalog with strong negation. For example, rules have a unique (`ID`) `ruleID` attribute in their `_rlab` element, so that superiority of one rule over the other can be expressed through an `IDREF` attribute of the superior rule. For example, the following rule `r2` is superior to rule `r`, presented above.

```
(defeasiblerule r2
   (declare (superior r1)) (visiting (name ?X))
⇒
   (not (tenured (name ?X))))
```

In RuleML notation, there is a `superiority` attribute in the rule label.

```
<imp>
   <_rlab ruleID="r2" ruletype="defeasiblerule"
superior="r1">
      <ind>r2</ind>
   </_rlab>
...
</imp>
```

Classes and objects (facts) can also be declared in DR-DEVICE; however, the focus in this paper is the use of RDF data as facts. The input RDF file(s) are declared in the rdf_import attribute of the `rulebase` (root) element of the RuleML document. There exist two more attributes in the `rulebase` element: the `rdf_export` attribute that declares the address of the RDF file with the results of the rule program to be exported, and the `rdf_export_classes` attribute that declares the derived classes whose instances will be exported in RDF/XML format.

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. Multiple constraints in the rule body can be expressed through the logical operators: `_not`, `_and`, `_or`.

```
<!ENTITY % LABELs "IDREFS">
<!ENTITY % CLASSes "NMTOKENS">
<!ATTLIST _rlab
   ruleID ID #REQUIRED
   ruletype (strictrule|defeasiblerule|defeater)
             #REQUIRED
   superior %LABELs; #IMPLIED>
<!ENTITY % _calc.cont "(function+)">
<!ELEMENT calc %_calc.cont;>
<!ENTITY % _head.content " (calc?, (atom | neg))">
<!ENTITY % _body.content "(atom | neg | and | or)">
<!ENTITY % _fname.cont "(#PCDATA)">
<!ELEMENT fname %_fname.cont;>
<!ENTITY % pos_term "(ind | var | function)">
<!ELEMENT function (fname, (%pos_term;)*)>
<!ENTITY % term "(_not | %pos_term;)">
<!ELEMENT _not (ind | var)>
<!ELEMENT _or (%term;, (%term;)+)>
<!ELEMENT _and (%term;, (%term;)+)>
<!ENTITY % constraint "(_not | _or | _and)">
<!ENTITY % _slot.content "(ind | var | %constraint;)">
<!ENTITY % negurdatalog_include SYSTEM
   "http://www.ruleml.org/0.85/dtd/neg/negurdatalog.dtd">
%negurdatalog_include;
<!ATTLIST rulebase
      rdf_import CDATA #IMPLIED
      rdf_export_classes %CLASSes; #IMPLIED
      rdf_export CDATA #IMPLIED>
```

Fig. 2. RuleML syntax DTD of the VDR-DEVICE rule language.

The Deductive Rule Language of R-DEVICE

R-DEVICE has a powerful deductive rule language which includes features such as normal (ground), unground, and generalized path expressions over the objects, stratified negation, aggregate, grouping, and sorting, functions. The rule language supports a second-order syntax, where variables can range over classes and properties. However, second-order variables are compiled away into sets of first-order rules, using instantiations of the metaclasses. Users can define views which are materialized and, optionally, incrementally maintained by translating deductive rules into CLIPS production rules. Users can choose between an OPS5/CLIPS-like or a RuleML-like syntax. Finally, users can use and define functions using the CLIPS host language. R-DEVICE belongs to a family of previous such deductive object-oriented rule languages (Bassiliades et al. 2000). Examples of rules are given below.

R-DEVICE, like DR-DEVICE, has both a native CLIPS-like syntax and a RuleML-compatible syntax. Here we will present a few examples using the former, since it is more concise. For example, assume there is an RDF class `carlo:owner` that defines the owners of the apartments and a property `carlo:has-owner` that relates an apartment to its owner.

The following rule returns the names of all apartments owned by "Smith":

```
(deductiverule test1
  (carlo:apartment (carlo:name ?x)
    ((carlo:lastName carlo:has-owner) "Smith"))
 =>
  (result (apartment ?x)))
```

The above rule has a ground path expression (`carlo:lastName carlo:has-owner`) where the right-most slot name (`carlo:has-owner`) is a slot of the "departing" class `carlo:apartment`. Moving to the left, slots be-long to classes that represent the range of the predecessor slots. In this example, the range of `carlo:has-owner` is `carlo:owner`, so the next slot `carlo:lastName` has domain `carlo:owner`. The value expression in the above pattern (e.g. constant "Smith") actually describes a value of the left-most slot of the path (`carlo:lastName`). Notice that we have adopted a right-to-left order of attributes, contrary to the left-to-right C-like dot notation that is commonly assumed, because we consider path expressions as function compositions, if we assume that each property is a function that maps its domain to its range.

Another example that demonstrates aggregate function in R-DEVICE is the following rule, which returns the number of apartments owned by each owner:

```
(deductiverule test2
  (carlo:apartment (carlo:name ?x)
        ((carlo:lastName carlo:has-owner) ?o))
 =>
  (result (owner ?o) (apartments (count ?x))))
```

Function `count` is an aggregate function that returns the number of all the different instantiations of the variable `?x` for each different instantiation of the variable `?o`. There are several other aggregate functions, such as `sum`, `avg`, `list`, etc.

Translating Defeasible Logic Rules into Deductive Rules

The translation of defeasible rules into R-DEVICE rules is based on the translation of defeasible theories into logic programs through the well-studied meta-program of (Antoniou et al. 2000). However, instead of directly using the meta-program at run-time, we have used it to guide defeasible rule compilation. Therefore, at run-time only first-order rules exist.

Before going into the details of the translation we briefly present the auxiliary system attributes (in addition to the user-defined attributes) that each defeasibly derived object in DR-DEVICE has, in order to support our translation scheme:

- `pos`, `neg`: These numerical slots hold the proof status of the defeasible object. A value of 1 at the `pos` slot denotes that the object has been defeasibly proven; whereas a value of 2 denotes definite proof. Equivalent `neg` slot values denote an equivalent proof status for the negation of the defeasible object. A 0 value for both slots denotes that there has been no proof for either the positive or the negative conclusion.
- `pos-sup`, `neg-sup`: These attributes hold the rule ids of the rules that can *potentially* prove the object positively or negatively.
- `pos-over`, `neg-over`: These attributes hold the rule ids of the rules that have overruled the positive or the negative proof of the defeasible object. For example, in the rules r_1 and r_2 presented above, rule r_2 has a negative conclusion that overrides the positive conclusion of rule r_1. Therefore, if the condition of rule r_2 is satisfied then its rule id is stored at the `pos-over` slot of the corresponding derived object.
- `pos-def`, `neg-def`: These attributes hold the rule ids of the rules that can defeat overriding rules when the former are superior to the latter. For example, rule r_2 is superior to rule r_1. Therefore, if the condition of rule r_2 is satisfied then its rule id is stored at the `neg-def` slot of the corresponding derived object along with the rule id of the defeated rule r_1. Then, even if the condition of rule r_1 is satisfied, it cannot overrule the negative conclusion derived by rule r_2 (as it is suggested by the previous paragraph) because it has been defeated by a superior rule.

Each *defeasible rule* in DR-DEVICE is translated into a set of 5 R-DEVICE rules:

- A *deductive* rule that generates the derived defeasible object when the condition of the defeasible rule is met. The proof status slots of the derived objects are initially set to 0. For example, for rule r_2 the following deductive rule is generated:

```
(deductiverule r2-deductive
  (visiting (name ?X))
⇒
  (tenured (name ?X) (pos 0) (neg 0)))
```

  Rule `r2-deductive` states that if an object of class `visiting` with slot `name` equal to `?X` exists, then create a new object of class `tenured` with a slot `name` with value `?X`. The derivation status of the new object (according to defeasible logic) is unknown since both its positive and negative truth status slots are set to 0. Notice that if a `tenured` object already exists with the same name, it is not created again. This is ensured by the value-based semantics of the R-DEVICE deductive rules.
- An aggregate attribute "*support*" rule that stores in `-sup` slots the rule ids of the rules that can potentially prove positively or negatively the object. For example, for rule r_2 the following "support" rule is generated (`list` is an aggregate function that just collects values in a list):

```
(aggregateattrule r2-sup
  (visiting (name ?X))
  ?gen23 <- (tenured (name ?X))
 ⇒
  ?gen23 <- (tenured (neg-sup (list r5))))
```

Rule `r2-sup` states that if there is a `visiting` object named `?X`, and there is a `tenured` object with the same name, then derive that rule `r2` could potentially support the defeasible negation of the `tenured` object (slot `neg-sup`).

- A derived attribute "*defeasibly*" rule that defeasibly proves either positively or negatively an object by storing the value of 1 in the `pos` or `neg` slots, if the rule condition has been at least defeasibly proven, if the opposite conclusion has not been definitely proven and if the rule has not been overruled by another rule. For example, for rule `r2` the following "defeasibly" rule is generated:

```
(derivedattrule r2-defeasibly
(visiting (name ?X) (pos ?gen29&:(>= ?gen29 1)))
  ?gen23 <- (tenured (name ?X) (pos ~2)
  (neg-over $?gen25&:(not (member$ r5 $?gen25))))
 ⇒
  ?gen23 <- (tenured (neg 1)))
```

Rule `r2-defeasibly` states that if it has been defeasibly proven that a `visiting` object named `?X` exists, and there is a `tenured` object with the same name that is not already strictly-positively proven and rule r_2 has not been overruled (check slot `neg-over`), then derive that the `tenured` object is defeasibly-negatively proven.

- A derived attribute "*overruled*" rule that stores in `–over` slots the rule id of the rule that has overruled the positive or the negative proof of the defeasible object, along with the ids of the rules that support the opposite conclusion, if the rule condition has been at least defeasibly proven, and if the rule has not been defeated by a superior rule. For example, for rule r_1 the following "overruled" rule is generated (through `calc` expressions, arbitrary user-defined calculations are performed):

```
(derivedattrule r1-over
  (professor (name ?X)
    (pos ?gen22&:(>= ?gen22 1)))
  ?gen16 <- (tenured (name ?X) (neg-sup $?gen19)
              (neg-over $?gen20)
              (pos-def $?gen18&
                :(not (member$ r4 $?gen18))))
 ⇒
(calc (bind $?gen21
  (create$ r1-over $?gen19 $?gen20)))
  ?gen16 <- (tenured (neg-over $?gen21)))
```

Rule `r1-over` actually overrules all rules that can support the negative derivation of `tenured`, including rule r_2. Specifically, it states that if it has been defeasibly proven that a `professor` object named `?X` exists, and there is a `tenured` object with the same name that its negation can be potentially supported by rules in the slot `neg-sup`, then derive that rule `r1-over` overruled those "negative supporters" (slot `neg-over`), unless it has been defeated (check slot `pos-def`).

- A derived attribute "*defeated*" rule that stores in `-def` slots the rule id of the rule that has defeated overriding rules (along with the defeated rule ids) when the former is superior to the latter, if the rule condition has been at least defeasibly proven. A "defeated" rule is generated only for rules that have a superiority relation, i.e. they are superior to others. For example, for rule r_5 the following "defeated" rule is generated:

```
(derivedattrule r2-def
  (visiting (name ?X)
    (pos ?gen29&:(>= ?gen29 1)))
  ?gen23 <- (tenured (name ?X) (pos-def $?gen26))
⇒
(calc (bind $?gen25 (create$ r2-def r1 $?gen26)))
  ?gen23 <- (tenured (pos-def $?gen25)))
```

Rule `r2-def` actually defeats rule r_1, since r_2 is superior to r_1. Specifically, it states that if it has been defeasibly proven that a `visiting` object named `?X` exists, and there is a `tenured` object with the same name then derive that rule `r2-def` defeats rule r_1 (slot `pos-def`).

Strict rules are handled in the same way as defeasible rules, with an addition of a derived attribute rule (called *definitely* rule) that definitely proves either positively or negatively an object by storing the value of 2 in the `pos` or `neg` slots, if the condition of the strict rule has been definitely proven, and if the opposite conclusion has not been definitely proven. For example, for the strict rule r_3: `visiting(X) → professor(X)`, the following "definitely" rule is generated:

```
(derivedattrule r3-definitely
  (visiting (name ?X) (pos 2))
  ?gen9 <- (professor (name ?X) (pos ~2))
  ⇒
  ?gen9 <- (professor (pos 2)))
```

Defeaters are much weaker rules that can only overrule a conclusion. Therefore, for a defeater only the "overruled" rule is created, along with a deductive rule to allow the creation of derived objects, even if their proof status cannot be supported by defeaters.

Execution Order

The order of execution of all the above rule types is as follows: "deductive", "support", "definitely", "defeated", "overruled", "defeasibly". Moreover, rule priority for stratified defeasible rule programs is determined by stratification. Finally, for non-stratified rule programs rule execution order is not determined. However, in order to ensure the correct result according to the defeasible logic theory for each derived attribute rule of the rule types "definitely", "defeated", "overruled" and "defeasibly" there is an opposite "truth maintenance" derived attribute rule that undoes (retracts) the conclusion when the condition is no longer met. In this way, even if rules are not executed in the correct order, the correct result will be eventually deduced because conclusions of rules that should have not been executed can be later undone. For example, the following rule undoes the "defeasibly" rule of rule r_2 when either the condition of the defeasible rule is no longer defeasibly satisfied, or the opposite conclusion has been definitely proven, or if rule r_5 has been overruled.

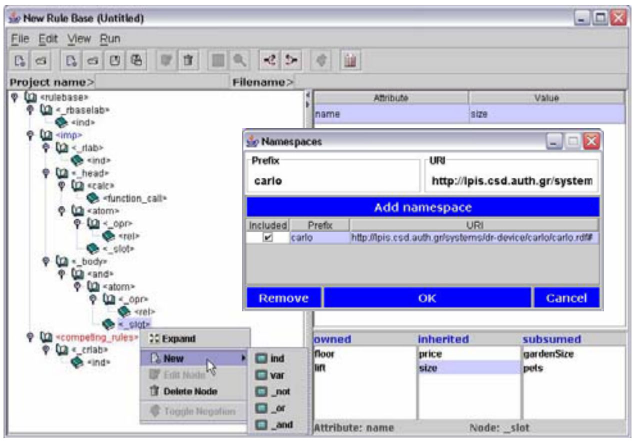
```
(derivedattrule r2-defeasibly-dot
  ?gen23 <- (tenured (name ?X) (neg 1)
                (neg-sup $? r5 $?))
  (not (and (visiting (name ?X) (pos ?gen29&
                 :(>= ?gen29 1)))
  ?gen23 <- (tenured (pos ~2) (neg-over $?g&
               :(not (member$ r2 $?g))))))
⇒
  ?gen23 <- (tenured (neg 0)))
```

DR-DEVICE has been tested for correctness using a tool that generates scalable test defeasible logic theories that comes with Deimos, a query answering defeasible logic system (Maher et al. 2001).

The Rule Editor

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 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.



```
<rulebase>
  <_rbaselab>
    <ind>
  <imp>
    <_rlab>
      <ind>
    <_head>
      <calc>
        <function_call>
      <atom>
        <_opr>
          <rel>
        <_slot>
    <_body>
      <and>
        <atom>
          <_opr>
            <rel>
          <_slot>
  <competing_rules>
    <_crlab>
      <ind>
```

Attribute	Value
name	size

Prefix	URI
carlo	http://lpis.csd.auth.gr/system

Included	Prefix	URI
✓	carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#

owned	inherited	subsumed
floor	price	gardenSize
lift	size	pets

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

More specifically, and as can be observed in Fig. 3, 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 in 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 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.

The right panel shows a table, which contains the attributes that correspond 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.

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 procedure behind tree expansion, which is "launched" 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 alternative 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. 3).

Another important component is the namespace dialog window (Fig. 3), where the user can determine which RDF/XML namespaces will be used by the rule base. Actually, we treat namespaces as addresses of input RDF Schema ontologies that contain the vocabulary for the input RDF documents, over which the rules 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 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.

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* (Maher et al. 2001) is a flexible, query processing system based on Haskell. It implements several variants, but neither conflicting literals nor negation as failure in the object language. Also, the current implementation does not integrate with Semantic Web, since it is solely a defeasible logic engine (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, although external translation modules could provide such interoperability. Finally, it is propositional and does not support variables.

Delores (Maher et al. 2001) 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 as well.

SweetJess (Grosof, Gandhe and Finin 2002) is yet another implementation of a defeasible reasoning system 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 VDR-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms). This applies to VDR-DEVICE to a large extent; however, the basic R-DEVICE language (Bassiliades and Vlahavas 2006) 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, VDR-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules (Bassiliades, Antoniou and Vlahavas 2006).

Mandarax (Dietrich et al. 2003) 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 (Bassiliades and Vlahavas 2006). Furthermore, DR-DEVICE supports both negation-as-failure and strong negation, and supports both deductive and defeasible logic rules.

Conclusions and Future Work

In this paper we have argued that defeasible reasoning is useful for many applications in the Semantic Web, mainly due to conflicting rules and rule priorities. We have also presented a system for defeasible reasoning on the Web, called VDR-Device. It is a visual environment for developing defeasible logic rule bases that, after analyzing the input RDF ontologies, it constrains the allowed vocabulary. Furthermore, the system employs a user-friendly graphical shell and a powerful defeasible reasoning system that supports the following:

- Multiple rule types of defeasible logic, such as strict rules, defeasible rules, and defeaters.
- Priorities among rules.
- Two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals.
- Compatibility with RuleML, the main standardization effort for rules on the Semantic Web.
- Direct import from the Web and processing of RDF data and RDF Schema ontologies.
- Direct export to the Web of the results (conclusions) of the logic program as an RDF document.

The defeasible reasoning system is built on-top of a CLIPS-based implementation of deductive rules. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes. However, the implementation is declarative because it interprets the not operator using Well-Founded Semantics.

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. We will try to visualize the semantics of defeasible logic in an intuitive manner, by providing graphical representations of rule attacks, superiorities, conflicting literals, 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. Finally, concerning the implementation of the graphical editor we will adhere to newer XML Schema-based versions of RuleML.

References

Antoniou, G. 1997. *Nonmonotonic Reasoning*. MIT Press.

Antoniou G. 2002. Nonmonotonic Rule Systems on Top of Ontology Layers. In *Proceedings of the 1st Int. Semantic Web Conference*. 394-398. LNCS 2342. Springer-Verlag.

Antoniou, G., and Arief, M. 2002. Executable Declarative Business Rules and their Use in Electronic Commerce. In *Proceedings of ACM Symposium on Applied Computing*. 6-10. ACM Press.

Antoniou, G., Billington, D., Governatori, G., Maher M.J. 2000. A Flexible Framework for Defeasible Logics. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*. 405-410. AAAI/MIT Press.

Antoniou, G., Billington, D., and Maher, M.J. 1999. On the Analysis of Regulations using Defeasible Rules. In *Proceedings of the 32nd Hawaii Int. Conference on Systems Science*, 7 pages (no page numbers). IEEE Press.

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

Antoniou, G., Skylogiannis, T., Bikakis, A., Bassiliades, N. 2005. DR-BROKERING – A Defeasible Logic-Based System for Semantic Brokering. In *Proceedings of IEEE Int. Conf. on E-Technology, E-Commerce and E-Service*. 414-417. IEEE Computer Society.

Ashri, R., Payne, T., Marvin, D., Surridge, M., and Taylor, S. 2004. Towards a Semantic Web Security Infrastructure. In *Proceedings of Semantic Web Services 2004 Spring Symposium Series*. Stanford University, Stanford California.

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

Bassiliades, N., and Vlahavas, I. 2006. R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata, *International Journal on Semantic Web and Information Systems*, 2(2) (to appear).

Bassiliades, N., Vlahavas, and I., Elmagarmid, A.K. 2000. E DEVICE: An extensible active knowledge base system with multiple rule type support. *IEEE TKDE*. 12(5): 824-844.

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

Boley, H., Tabet, S., and Wagner, G. 2001. Design Rationale for RuleML: A Markup Language for Semantic Web Rules. *SWWS 2001*: 381-401.

Dean, M., and Schreiber, G. eds. 2004. *OWL Web Ontology Language Reference*. `www.w3.org/TR/2004/REC-owl-ref-20040210/`

Dietrich, J.; Kozlenkov A.; Schroeder, M.; Wagner, G. 2003. Rule-based agents for the semantic web. *Electronic Commerce Research and Applications*. 2(4):323–338.

Governatori, G. 2005. Representing business contracts in RuleML, *International Journal of Cooperative Information Systems*, 14(2-3):181-216.

Governatori, G., Dumas, M., Hofstede, A. ter and Oaks P. 2001. A formal approach to protocols and strategies for (legal) negotiation, In *Proceedings of the 8th International Conference of Artificial Intelligence and Law*. 168-177. ACM Press.

Grosof, B.N., Gandhe, M.D., Finin, T.W. 2002. SweetJess: Translating DAMLRuleML to JESS. In *Proceedings of Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*. Held at 1st Int. Semantic Web Conference.

Grosof, B. N. and Poon T. C. 2003. SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings of the 12th Int. Conference on World Wide Web*. 340-349. ACM Press.

Hayes, P., “RDF Semantics”, *W3C Recommendation*, Feb. 2004, `http://www.w3c.org/TR/rdf-mt/`

Maher, M.J.; Rock, A.; Antoniou, G.; Billington, D.; Miller T. 2001. Efficient Defeasible Reasoning Systems. *Int. Journal of Tools with Artificial Intelligence*. 10(4):483-501.

Marek, V.W., Truszczynski, M. 1993. *Nonmonotonic Logics; Context Dependent Reasoning*. Springer-Verlag.

McBride, B. 2001. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the 2nd Int. Workshop on the Semantic Web*, CEUR Workshop Proceedings, Vol. 40.

Nute, D. 1987. Defeasible Reasoning. In *Proceedings of the 20th Int. Conference on Systems Science*, 470-477. IEEE Press.