A Graphical Rule Authoring Tool for Defeasible Reasoning in the Semantic Web

Nick Bassiliades[1], Efstratios Kontopoulos[1], Grigoris Antoniou[2], and Ioannis Vlahavas[1]

[1]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
{nbassili, skontopo, vlahavas}@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. However, the syntax of defeasible logic may appear too complex for many users. In this paper we present a graphical authoring tool for defeasible logic rules that acts as a shell for the DR-DEVICE defeasible reasoning system over RDF metadata. The tool helps users to develop a rule base using the OO-RuleML syntax of DR-DEVICE rules, by constraining the allowed vocabulary through analysis of the input RDF namespaces, so that the user does not have to type-in class and property names. Rule visualization follows the tree model of RuleML. The DR-DEVICE reasoning system is implemented on top of the CLIPS production rule system and builds upon an earlier deductive rule system over RDF metadata that also supports derived attribute and aggregate attribute rules.

1. Introduction

The development of the Semantic Web [8] proceeds in layers, each layer being on top of other layers. At present, the highest layer that has reached sufficient maturity is the ontology layer, with OWL [11], a description logic based language, being the standard. The next step in the development of the Semantic Web will be the logic and proof layers and rule systems appear to lie in the mainstream of such activities. 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; and (b) they can be used to develop declarative systems on top of (using) ontologies.

Defeasible reasoning is a simple rule-based approach to reasoning with incomplete and inconsistent information. It can represent facts, rules, and priorities among rules. This reasoning family comprises defeasible logics ([3]) and Courteous Logic Programs [14]. The main advantage of this approach is the combination of two desirable features: a) enhanced representational capabilities, allowing one to reason with in-

complete and contradictory information, coupled with b) low computational complexity compared to mainstream nonmonotonic reasoning.

Defeasible logic can easily express 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 ([5], [16]), business rules [1], personalization, brokering [4], bargaining, and automated agent negotiations ([13], [19]).

However, defeasible logic is certainly not an end-user language but rather a developer's one, because its syntax may appear too complex. In this paper, we present a graphical rule authoring tool for defeasible logic that acts as a shell for the DR-DEVICE [6] defeasible reasoning system for the Semantic Web. This rule authoring tool is built in Java and helps users to develop a rule base using the OO-RuleML [9] syntax of DR-DEVICE rules. Among others, the tool constrains the allowed vocabulary, by analyzing the input RDF namespaces; 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.

DR-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 [9] 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 [10] implementation of deductive rules, namely the R-DEVICE system [7]. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes.

The paper is organized as follows. Section 2 briefly introduces the syntax and semantics of defeasible logics. Section 3 presents the architecture of the DR-DEVICE reasoning system. Section 4 describes the RuleML syntax of defeasible logic rules in DR-DEVICE. Section 5 presents the graphical rule authoring tool. Finally, section 6 discusses related work, and section 7 concludes the paper with a summary and description of current and future work.

2. An Introduction to Defeasible Logics

A *defeasible theory D* is a couple $(R,>)$ where R a finite set of rules and $>$ a superiority relation on R. Each rule has a unique rule label. There are three kinds of rules: strict rules, defeasible rules, and defeaters.

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 "*Penguins are birds*". Written formally: r_1: penguin(X) $\rightarrow$ bird(X). Inference from strict rules only is called *definite inference*. Strict rules are intended to define relationships that are definitional in nature and such an example is ontological knowledge.

Defeasible rules are denoted by $A \Rightarrow p$, and can be defeated by contrary evidence. An example of such a rule is `r2: bird(X) ⇒ flies(X)`, which reads as: "*Birds typically fly*".

Defeaters are denoted as $A \sim> p$ and are not used to actively support conclusions, but only to prevent some of them. An example of such a defeater is: `r4: heavy(X) ~> ¬flies(X)`, which reads as: "*Heavy birds may not fly*".

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 r_2 and r_3: `penguin(X) => ¬flies(X)`, no conclusive decision can be made about whether a penguin flies, because rules r_2 and r_3 contradict each other. But if we introduce a superiority relation > with $r_3 > r_2$, then we can indeed conclude that a penguin does not fly.

3. The DR-DEVICE System Architecture and Functionality

The DR-DEVICE reasoning system consists of two major components (Fig. 1): the RDF loader/translator and the rule loader/translator. The user submits to the rule loader a rule program (a URL or a local file name) that contains a) one or more rules in RuleML-like syntax [9], 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 RDF output document.

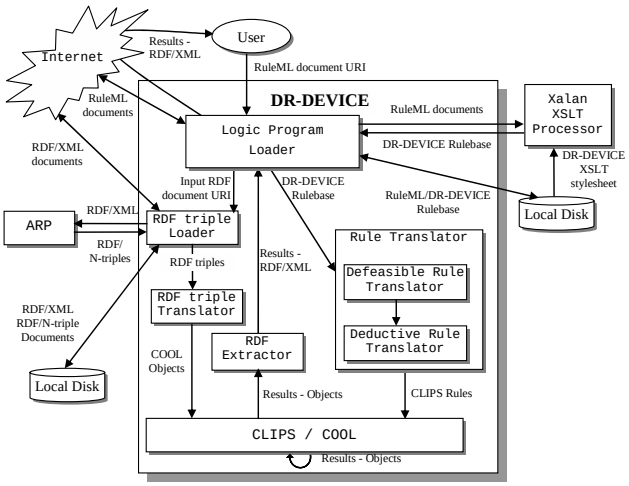

Fig. 1. Architecture of the DR-DEVICE reasoning system.

The RuleML file is transformed into the native CLIPS-like syntax through an XSLT stylesheet. The DR-DEVICE rule program is then forwarded to the rule translator. 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 of R-DEVICE [7].

The *rule translator* compiles the defeasible logic rules into a set of CLIPS production rules in two steps: First, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic R-DEVICE rule language, using the translation scheme described in [6]. Then, all these R-DEVICE rules are translated into CLIPS production rules [10], according to the R-DEVICE rule translation scheme [7]. All compiled rule formats are kept into local files, so that the next time they are needed they can be directly loaded, improving speed.

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 or query. 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 RDF Schema definitions for the exported derived classes and the instances of the exported derived classes, which have been proven (positively or negatively, defeasibly or definitely).

4. The Defeasible Logic Language of DR-DEVICE

DR-DEVICE has both a native CLIPS-like syntax ([6]) and a RuleML-compatible syntax [9]. Rules are encoded in an `imp` element and they have a label (`_rlab` element), which also includes the rule's unique ID (`ruleID` attribute) and the type of the rule (`ruletype` attribute). The names (`rel` elements) of the operator (`_opr`) elements of atoms are class names, since atoms actually represent CLIPS objects. Atoms have named arguments, called slots, which correspond to object properties. In DR-DEVICE, RDF resources are represented as CLIPS objects; therefore, atoms correspond to queries over RDF resources of a certain class with certain property values. For example, the following fragment represents the defeasible rule r_2 of section 2:

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

Superiority relations are represented as attributes of the superior rule. For example, rule r_3 (of section 2) is superior to rule r_2. In RuleML, this is represented via a `superiority` attribute in the rule label of rule r_3. Negation is represented via a `neg` element that encloses an `atom` element.

```
<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>
```

Apart from the rule declarations, there are `comp_rules` elements that declare groups of competing rules which derive competing positive conclusions, also known as *conflicting literals*.

```
<comp_rules c_rules="r10 r11 r12">
  <_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, in the header of the rulebase several important parameters are declared; 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, which is the RDF file with the exported results, and the `rdf_export_classes` attribute, which are the derived classes, whose instances will be exported in RDF/XML format. An example is shown below:

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

5. The Graphical Rule Authoring Tool

As the previous section shows, expressing or even viewing rules in RuleML syntax can often be highly cumbersome. In order to enhance user-friendliness and efficiency, DR-DEVICE is supported by a Java-built graphical authoring tool, which also acts as a graphical shell for the DR-DEVICE core reasoning system.

The graphical shell facilitates the development and invocation of rulebases, by calling the external applications that constitute the DR-DEVICE system. Users can evoke local or remote RuleML rulebases by starting new projects. The rulebase is then displayed in the left frame in XML-tree format, also offering the user the capability of navigating through the entire tree (Fig. 2). When the rulebase is compiled and run, the DR-DEVICE core system, described in section 3, is evoked. The execution trace is watched via a DOS Window which can be later re-examined using the 'Run Trace' window (Fig. 3). Users can set the level of detail during the trace, using the Parameters menu. The exported results of the inference process can be examined via an Internet Explorer window (Fig. 3). Finally, users can also re-run already compiled projects considerably (10-times) faster.

The graphical authoring tool facilitates rulebase developers via constrained yet flexible deployment of pre-defined rule templates, according to both the RuleML-compatible syntax and the RDF-oriented semantics.

While traversing the XML tree, the user can add or remove elements, according to the DTD limitations. In general, the rule editor allows only a limited number of opera-

tions performed on each element, according to the element's meaning within the rule tree. The main principle of tree expansion is the following: when a new element is added, then all the mandatory sub-elements are also added. When there are multiple alternative sub-elements, none is added, but the user can select one of them to add by right-clicking on the parent element. Furthermore, the user can alter the textual content (PCDATA) of the tree leafs. The `atom` element has a special treatment because it can be either negated or not. To facilitate this, the wrapping/unwrapping of an `atom` element within a `neg` element is performed via a toggle button.

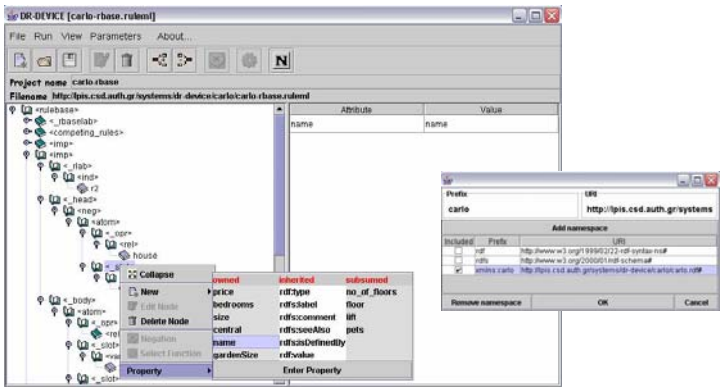

Fig. 2. The graphical rule authoring tool of DR-DEVICE and the namespace dialog window.

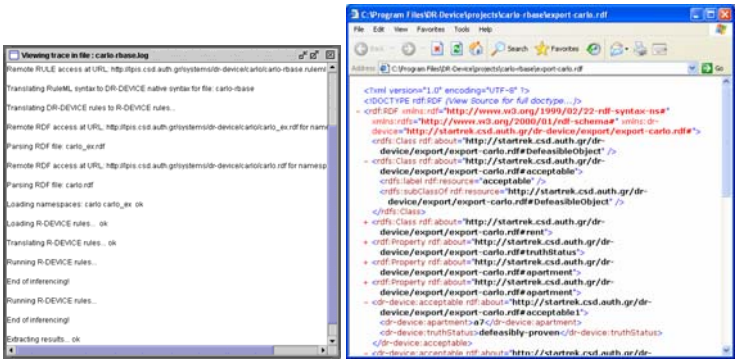

Fig. 3. The Run Trace and Results windows of the graphical DR-DEVICE shell.

The attribute editing area (in the right-hand frame) shows the XML attributes, corresponding to the selected tree node in the XML navigation-editing area. All the attributes (both mandatory and optional) are shown in this area, but the final XML document will contain only the non null ones. Rule IDs are treated specially, since they uniquely represent rules within the rulebase. Some IDREF attributes, such as the `superior` attribute of the `_rlab` element, use the list of rule IDs to constrain the list of possible values. Names of functions appearing in an `fcall` element are constrained to be among the system-defined CLIPS functions.

One of the important aspects of the rule editor is the namespace dialog window (Fig. 2), where the user can declare all the XML namespaces that will be used throughout the rulebase. Actually, namespace declarations are addresses of RDF Schema documents that contain the vocabulary for the input RDF documents over which the rules of the rulebase will be run. Namespaces are analyzed by the authoring tool in order to discover the allowed class and property names for the rulebase being edited. These names are used in pull-down menus and name lists throughout the authoring of the RuleML rulebase, in order to constrain the allowed names that can be used, to facilitate rule authoring and, consequently, reduce the possible semantical (and syntactical) errors of the rule developer.

More specifically, RDF Schema documents are parsed, using the ARP parser of Jena [18], a flexible Java API for processing RDF documents, and 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 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). This vector 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.

The vector CV_d is used for loosely suggesting possible values for `rel` elements of the `atom` in a rule head, but not constraining them, because rule heads can either introduce new derived classes or refer to already existing ones.

The *full class vector* (CV_f), which is a union of the above two vectors ($CV_f = CV_b \cup CV_d$), 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 parsed for property names and their domains. The properties 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`) $\rightarrow P \in PV_b$

Except from the base property vector, there also exists the *derived property vector* (PV_d), which contains the names of the properties of the derived classes, i.e. the properties of classes which lie at rule heads (conclusions). This vector is initially empty and is dynamically extended every 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: $PV_f = PV_b \cup PV_d$.

For each property P in the PV_f vector an object is created that maintains all the RDF Schema information needed. More specifically, each P object maintains two sets: *superproperty set SUPP*(P) and *domain set DOM*(P).

The *SUPP*(P) set initially contains all 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) $\rightarrow SP \in SUPP(P)$

The $SUPP(P)$ set is then further populated with the indirect superproperties of each property, by recursively traversing upwards the property hierarchy. Existing duplicates due to multiple inheritance are subsequently merged, since $SUPP(P)$ is a set:

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

The $DOM(P)$ set of domains is initially constructed, by examining the domain of each property declared in the RDF Schema documents. The domain of each derived class property is the corresponding derived class:

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

The domain of the RDF built-in properties BIP is `rdfs:Resource`:

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

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

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

The $DOM(P)$ set is then further populated, by inheriting the domains of all the superproperties (both direct and indirect):

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

This follows from the RDFS semantics, which dictate that the domains (and ranges) of the superproperties are inherited by the subproperties conjunctively.

The domains of the properties are needed, in order to constrain the possible values that the slot names can take, when editing the RuleML tree. More specifically, for each `atom` element appearing inside the rule body, when the class name 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. Furthermore, subsumed properties are also allowed, as it is explained below.

In order to link each class with the allowed properties, for each class C in the CV_f vector an object is created that maintains all the RDF Schema information needed. More specifically, each C object maintains five sets: *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:

$\forall C \in CV_b\ \forall SC \in CV_b$, ($C$ `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`:

$\forall C \in CV_b$, $C \neq$ `rdfs:Resource` $\wedge$ ($\neg\exists SC\ SC \in CV_b \rightarrow$ (C `rdfs:subClassOf` SC))
$\rightarrow$ `rdfs:Resource` $\in SUPC(C)$

Derived classes are considered to be subclasses of `rdfs:Resource`:

$\forall C \in CV_d$, `rdfs:Resource` $\in SUPC(C)$

The $SUPC(C)$ set is then further populated with the indirect superclasses of each class, by recursively traversing upwards the class hierarchy. Potential duplicates due to multiple inheritance are again subsequently merged, since $SUPC(C)$ is a set:

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

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

$\forall C \in CV_b \; \forall SC \in SUPC(C) \rightarrow C \in SUBC(SC)$

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):

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

This follows from the RDFS semantics, which dictate that the instances of a subclass are also instances of its superclass; therefore, properties which have the superclass as domain can also have any of its subclasses as domain.

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 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. 2), when the class of the atom element is C.

6. Related Work

There exist several previous implementations of defeasible logics. *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 [7] 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 [6].

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 [7]. Furthermore, DR-DEVICE supports both negation-as-failure and strong negation, and supports both deductive and defeasible logic rules.

7. Conclusions and Future Work

In this paper we argued that defeasible reasoning is useful for many applications in the Semantic Web, such as modeling policies and business rules, agent brokering and negotiation, ontology and knowledge merging, etc., mainly due to conflicting rules and rule priorities. However, the syntax of defeasible logic may appear too complex for many users; therefore, we have developed a graphical authoring tool to facilitate users in developing a rulebase using a RuleML-compatible defeasible logic rule language. DR-DEVICE is a defeasible reasoning system over RDF metadata, which is implemented on top of the CLIPS production rule system. The rule authoring tool constrains the allowed vocabulary by analyzing the input RDF namespaces, so that the user does not have to type-in class and property names, preventing potential syntactical and semantical errors. Rule visualization follows the tree model of RuleML.

In the future, we plan to delve into the proof layer of the Semantic Web architecture by developing further the graphical environment into a full visual IDE that includes 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.

Acknowledgments

This work is partially funded by the Greek Ministry of Education (EPEAEK) and the European Union under the Pythagoras II programme.

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., 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.

[5] 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.

[6] Bassiliades N., Antoniou, G., Vlahavas I., "A Defeasible Logic Reasoner for the Semantic Web", 3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004), Springer-Verlag, LNCS 3323, pp. 49-64, Hiroshima, Japan, 2004.

[7] Bassiliades N., Vlahavas I., "R-DEVICE: A Deductive RDF Rule Language", 3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004), Springer-Verlag, LNCS 3323, pp. 65-80, Hiroshima, Japan, 2004.

[8] Berners-Lee T., Hendler J., and Lassila O., "The Semantic Web", *Scientific American*, 284(5), 2001, pp. 34-43.

[9] Boley H., Tabet S., *The Rule Markup Initiative*, `www.ruleml.org/`

[10] *CLIPS Basic Programming Guide* (v. 6.21), `www.ghg.net/clips/CLIPS.html`

[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] 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.