International Journal on Artificial Intelligence Tools
Vol. 17, No. 5 (2008) 903–924
© World Scientific Publishing Company

VISUAL MODELING OF DEFEASIBLE LOGIC RULES WITH DR-VisMo

EFSTRATIOS KONTOPOULOS
Department of Informatics, Aristotle University of Thessaloniki,
Thessaloniki, GR-54124, Greece
skontopo@csd.auth.gr

NICK BASSILIADES
Department of Informatics, Aristotle University of Thessaloniki,
Thessaloniki, GR-54124, Greece
nbassili@csd.auth.gr

GRIGORIS ANTONIOU
Institute of Computer Science, Foundation for Research and Technology, Hellas (FORTH),
Heraklion, GR-71110, Greece
antoniou@ics.forth.gr

ANNA SERIDOU
Department of Informatics, Aristotle University of Thessaloniki,
Thessaloniki, GR-54124, Greece
aseridou@csd.auth.gr

The standardization of the Semantic Web has reached as far as ontologies and ontology languages. However, in order for the full potential of the Semantic Web to be achieved, the ability of reasoning over the available information is also essential. Rules can assist in this affair and various logics have been proposed for the Semantic Web domain. One of them is defeasible reasoning that deals with incomplete and conflicting information. However, despite its solid mathematical notation, it may be confusing to end users. To confront this downside, we proposed a representation schema for defeasible logic rule bases, which is based on directed graphs that feature distinct node and connection types. This paper presents DR-VisMo, a defeasible logic rule base editor and visualization system that implements this representation approach. The system also features a stratification algorithm for visualizing rule bases that deals with decisions, regarding the arrangement of the various elements in the graph. DR-VisMo is implemented as part of VDR-DEVICE, an environment for modeling and deploying defeasible logic rule bases on top of RDF ontologies.

Keywords: Semantic Web; defeasible reasoning; directed graphs; visualization.

1. Introduction

The standardization of the *Semantic Web*[1] has reached as far as ontologies and ontology languages, with OWL, the Web Ontology Language, being currently the leading standard in ontology representation. However, in order for the full potential of the Semantic Web to be achieved, the ability of reasoning over the information available in the Web is also essential, as stated by Tim Berners-Lee *et al.*[1] Rules can assist in this affair, by providing a well-known reasoning mechanism, with established theory and implementations.

Various logics have been proposed for the Semantic Web domain. One of them is *defeasible reasoning*,[2] a member of the non-monotonic reasoning family that represents a rule-based approach to reasoning with incomplete and conflicting information. It can represent facts, rules, priorities and conflicts among rules. Compared to mainstream non-monotonic reasoning, the main advantages of defeasible reasoning are enhanced representational capabilities[3] coupled with low computational complexity.[4]

Defeasible reasoning features a solid mathematical notation, which gives it credibility. However, the very same mathematical background may seem confusing to end users. *Directed graphs* (*digraphs*) can assist in confronting this drawback. They are a flexible visualization tool, offering a comprehensible way to represent relationships between entities.[5] Their applicability, however, is balanced by the fact that it is difficult to associate data of a variety of types with the nodes and with the connections between the nodes in the graph.

This paper presents DR-VisMo, a defeasible logic rule base editor and visualization system. The representation schema of the software is based on directed graphs and was presented in a previous work of ours.[6] By applying digraphs, we attempt to exploit their expressiveness, but also try to mitigate their main disadvantage, mentioned above, by proposing distinct node types for rules and atomic formulas and distinct connection types for each rule type in defeasible logic and for superiority relationships. DR-VisMo also features a stratification algorithm[7] for visualizing rule bases. The algorithm deals with decisions, regarding the arrangement of the various elements in the graph, a task that considerably improves clarity. Notice that stratification is solely used for visualization purposes and is indifferent, regarding the underlying defeasible logic inference engine, since rule cycles in defeasible logic (with the presence of strong negation) are treated skeptically and no conclusion is derived. The main contribution of the paper, nevertheless, involves the presentation of DR-VisMo as a whole, including its rule authoring module. DR-VisMo is implemented as part of VDR-DEVICE,[8] an environment for modeling and deploying defeasible logic rule bases on top of RDF ontologies.

The rest of the paper is organized as follows: Section 2 describes the key aspects of applying directed graphs for the representation of defeasible logic rules, emphasizing on the representation of arguments and conditions. The next section describes DR-VisMo, focusing on its two main functionalities, namely, the rule authoring and rule base visualization modules. Section 4 presents a user evaluation of the system, while the next section discusses related work, followed by the conclusions and ideas for future research.

2. Defeasible Logics and Digraphs

A defeasible theory D (i.e. a knowledge base or a program in defeasible logic) consists of three basic components: a set of facts (F), a set of rules (R) and a superiority relationship (>). Therefore, D can be represented by the triple (F, R, >).

The representation of defeasible logic rules in our approach is based on the methodology presented by Nute,[9] who applies *d-graphs* for visualizing a defeasible logic rule base. However, the method we adopt adds extra features to the graph that offer expressiveness. More specifically, each rule base in our approach is represented by an *oriented graph* G: = (V, A) (a directed graph with no bi-directed edges[5]), where:

- V is a set of vertices, with each vertex being either a rectangle that represents a literal and is called a "literal box", or a circle, representing a rule, and
- A is a set of arcs, formally defined $A \subseteq \{(x, y) \mid x, y \in V\}$, where each arc is directed from a graph element x to another graph element y, respectively. Similarly to Nute's approach,[9] arcs belong to several types: one for each rule type in defeasible logic (strict rules, defeasible rules and defeaters), one for superiority relationships, plus a fifth connection type, used for consistency purposes. More details are included in a subsequent section.

2.1. *Rule types in defeasible logic*

The full theoretical approach, regarding the graphical representation of defeasible reasoning elements has been thoroughly described;[6] here only a brief outline will be made. First of all, let us consider *Alice*, a music fan, who wants to create a music-related rule base. The first rule type in defeasible reasoning is *strict rules*, denoted by $A \rightarrow p$ and interpreted in the typical sense: whenever the premises are indisputable, then so is the conclusion. Thus, if Alice would like to express the statement: "*A hard rock song is a kind of rock song*", she would have to formalize the following strict rule: `r1: hard_rock(X) → rock(X)`, which is represented by the digraph in Fig. 1.

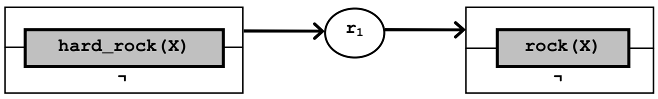


Fig. 1. Visual representation of strict rule r_1.

Each literal box consists of two adjacent (and conflicting) "*atomic formula boxes*", where the upper one represents a positive and the lower one a negated atomic formula. This way, these two conflicting, but also related, atomic formulas are depicted together distinctively, maintaining their independence. Notice also that for the sake of presentation clarity we currently only represent the predicate and not the literal (i.e. predicate plus all the arguments). Nevertheless, the full representation (presented later) includes a full-fledged representation of literals.

Defeasible rules, on the other hand, can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. Two examples are: r_2: rock(X) ⇒ likes(X) ("*Alice usually likes rock songs*") and r_3: hard_rock(X) ⇒ ¬likes(X) ("*Alice typically does not like hard rock songs*"). Both are depicted in Fig. 2.

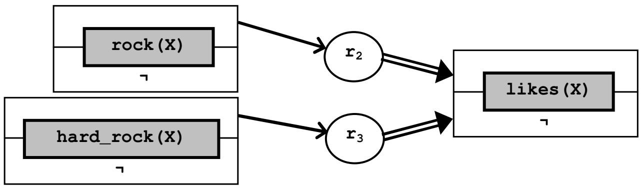


Fig. 2. Representing defeasible rules r_2 and r_3.

Defeaters, denoted by *A ~> p*, do not actively support conclusions, but can only prevent some of them. If Alice, for example, would like to express the fact that she may not like cover versions of rock songs, she would have to formalize the defeater: r_2': cover(X) ~> ¬likes(X). This defeater can defeat, for example, rule r_2 mentioned above and it can be represented by Fig. 3. Rule r_2' actually introduces ambiguity regarding cover songs and Alice's preferences, which should be resolved through other rules. However, the defeater alone cannot actively support the conclusion that Alice does not like a song, simply because it is a cover version.

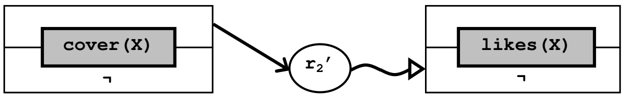


Fig. 3. Visual representation of defeater r_2'.

Finally, the *superiority relationship* among the rule set R is an acyclic relation > on R, used, in order to resolve conflicts among rules. For example, given the defeasible rules r_2 and r_3 above, no conclusive decision can be made about whether Alice does like hard rock music or not, because rules r_2 and r_3 contradict each other. But if the superiority relationship $r_3 > r_2$ is introduced, then r_3 overrides r_2 and we can indeed conclude that Alice does not like hard rock. In this case, rule r_3 is called *superior* to r_2 and r_2 *inferior* to r_3. A fourth connection type is introduced for superiority relationships, which is displayed in Fig. 4.

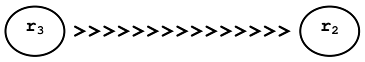


Fig. 4. Visual representation of $r_3 > r_2$.

According to defeasible logic proof theory,[10] in order to show that q is provable defeasibly there are two choices: (1) to show that q is already definitely provable, using a strict rule; or (2) to show that there is a strict or defeasible rule with head q whose body literals have been defeasibly proven and there are no possible “attacks”, that is, reasoning chains in support of ¬q. Formally, we must show that ¬q is not definitely provable. Also we must consider the set of all rules which are not known to be inapplicable and which have head ¬q (here we consider defeaters, too, whereas they could not be used to support the conclusion q). Essentially each such rule attacks the conclusion q. For q to be provable, each attacker must be counterattacked by another rule with head q with the following properties: (i) the counter-attacker must be applicable, and (ii) it must be stronger than (i.e. superior to) the attacker. Thus each attack on the conclusion q must be counterattacked by a stronger rule.

2.2. *Representing arguments and conditions*

So far we have shown how rules are represented by interconnecting literal boxes with rule nodes. However, we have not yet included how literal arguments are presented, either being variables or constants. Also, variables are usually associated with simple conditions, such as Y>=1960, which could be represented as predicates, but it is practically more convenient to consider them more closely related to the closest literal that contains the corresponding variable as an argument.

Arguments are incorporated inside the literal box just after the predicate name. The set of all arguments for each literal box is called *argument pattern*. For instance, the literal `year(X,2000)`, which could state that the year a song X was released is 2000, is represented as in Fig. 5 (a). Simple conditions associated with any of the variables of a literal can also appear inside the literal box, each on a separate line (called *condition pattern*) below the literal. For example, if the fragment `year(X,Y),Y>=1960` appears in a rule condition, it can be represented as in Fig. 5 (b).

A certain predicate, say `year`, can appear many times in a rule base, in rule conditions or even rule conclusions. All literal boxes of the same predicate can be grouped, so that the user can realise that all these boxes refer to the same set of literals. To this end, we introduce the notion of a *predicate box*, a container for all literal boxes that refer to the same predicate. The literal boxes inside the predicate box "share" the predicate name that is located at the top of the predicate box. This approach is a temporary convention, needed in order to introduce the complete representation in the following sections. The literal boxes inside predicate boxes that express conditions on instances of the specific predicate extension are called *predicate patterns*. For example, the literal boxes of Fig. 5 can be grouped inside a predicate box as in Fig. 6. Notice that each predicate pattern contains exactly one argument pattern and zero, one or more condition patterns.

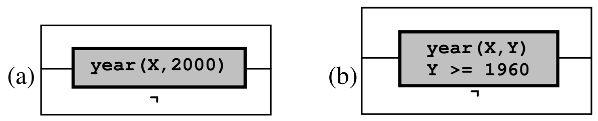


Fig. 5. Representing (a) arguments of literals and (b) simple conditions on variables.

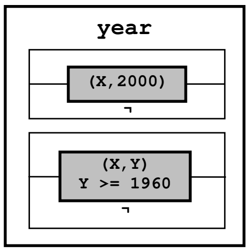


Fig. 6. Predicate box and predicate patterns.

3. DR-VisMo: Defeasible Reasoning — Visualizing and Modeling

DR-VisMo is a visual rule editor that assists users in modeling and visualizing defeasible logic rule bases. It is implemented as part of the *VDR-DEVICE*[8] system, an integrated development environment for deploying defeasible logic rule bases on top of RDF ontologies.

The core component of VDR-DEVICE is DR-DEVICE,[11] a reasoning system that processes RDF data, performs the defeasible inference procedure, produces the results and exports them as RDF data. The reasoning system employs an object-oriented RDF data model, which is different from the established triple-based RDF data model, treating properties as typical encapsulated attributes of resource objects. This way, properties of resources are not scattered across several triples, as in most other RDF inference systems, increasing query performance due to fewer joins.[12]

DR-DEVICE rule bases are expressed in an extension[11] of RuleML. Extensions deal with two aspects of DR-DEVICE, namely defeasible logic and its CLIPS[13] implementation. Defeasible logic extensions include rule types, superiority relations and conflicting literals, while CLIPS-related extensions deal with constraints on predicate arguments and functions.

A fragment of a rule is displayed in Fig. 7. The names (`rel` elements) of the operator (`_opr`) elements of atoms are class names, since atoms actually represent CLIPS objects.[13] RDF class names, used as base classes in the rule condition, are referred to via the `href` attribute of the `rel` element (e.g. `hard_rock` in Fig. 7, which responds to hard rock songs), while derived class names are text values of the `rel` element. Atoms have named arguments (slots), which correspond to object/RDF properties. Since RDF resources are represented as CLIPS objects, atoms in the rule body correspond to queries over RDF resources of a certain class with certain property values, while atoms in the rule head correspond to templates of materialized derived objects, which are exported as RDF resources at the end of the inference process.[11,12]

The following two sections describe the processes of developing (section 3.1) and visualizing (section 3.2) a defeasible logic rule base with the help of DR-VisMo, while section 3.3 presents the system architecture and functionality.

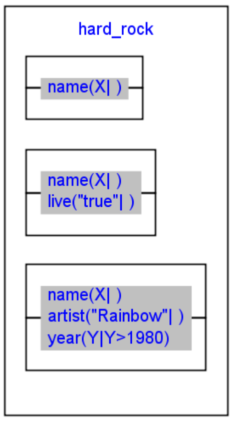
3.1. *Rule base development*

The rule base graph consists of a variety of elements. Initially, for each class that the user wants to be created, a *class box* with the same name is constructed. Class boxes are the equivalent of predicate boxes, described previously, and they are populated with one or more *class patterns*, the equivalent of predicate patterns.

In practice, class patterns express selection conditions over instances of the specific class. Visually, class patterns appear as literal boxes. This mapping is justified by the fact that atoms – expressed in the RuleML-like language of VDR-DEVICE – are actually atomic formulas (they correspond to queries over RDF resources of a certain class with certain property values). Thus, the truth value associated with each returned class instance will be either positive or negative.

Similarly to class boxes, class patterns are populated with one or more *slot patterns*, which are the equivalent of *argument* and *condition patterns*. There are, however, certain differences that arise from the different nature of the tuple-based model of predicate logic and the object-based model of VDR-DEVICE. In the latter, class instances are queried via named slots rather than positional arguments. Not every slot needs to be queried and the position of the slot inside the object is irrelevant. Therefore, instead of a single-line argument pattern there is a set of slot patterns in many lines; each slot pattern is identified by the slot name. Furthermore, in the VDR-DEVICE RuleML-like syntax, simple conditions are attached to the slot patterns; this is reflected to the visual representation where condition patterns are encapsulated inside the associated slot patterns.

An example of all the above is seen in Fig. 7, which shows a class box (created with DR-VisMo) that contains three class patterns applied on the *hard_rock* class and a code fragment matching the third class pattern, written in the RuleML-like syntax of VDR-DEVICE. The class patterns contain 1, 2 and 3 slot patterns respectively. The argument list of each slot pattern is divided in two parts, separated by a colon; the variable is placed on the left and the corresponding expressions and conditions are placed on the right. The



```
<atom>
  <_opr>
    <rel href="hard_rock"/>
  </_opr>
  <_slot name="name">
    <ind>x</ind>
  </_slot>
  <_slot name="artist">
    <ind>"Rainbow"</ind>
  </_slot>
  <_slot name="year">
    <_and>
      <var>y</var>
      <function_call name="&gt;">
        <var>y</var>
        <ind>1980</ind>
      </function_call>
    </_and>
  </_slot>
</atom>
```

Fig. 7. A class box example and a code fragment for the second class pattern.

variable in the slot pattern is used, in order for the slot value to be unified, with the latter having to satisfy the list of constraints. In other words, slot patterns represent conditions on slots (or class properties). In the case of constant values, only the left-hand side is utilized; the second and third class patterns, for instance, contain such examples. To sum things up, the first class pattern represents a query on all instances of the *hard_rock* class that have a name, i.e. all the named hard-rock songs; the second class pattern queries all the named hard-rock songs that were performed "live", while the third one represents a query on all the hard-rock songs by the "Rainbow" band that were released after 1980.

Besides class boxes, class patterns and slot patterns, users can also create *rule circles* that represent rules and *arcs* that connect the nodes in the graph. Rule circles contain the unique rule ID assigned by the user and their appearance was described in a previous section. As for the connections in the graph, there exist five types of them, as stated earlier: three for the rule type (strict, defeasible, defeater), one for the superiority relationship, plus a simple arrow connection type for connecting the class patterns of rule bodies to the rule circles. A sample rule graph, containing several of the features described above can be seen in Fig. 9.

3.2. *Rule base visualization*

Besides modeling defeasible logic rule bases, DR-VisMo can also visualize an existing rule base. The first step involves collecting the class names.

3.2.1. *Collecting the class names*

The RDF Schema documents, designated by the user, are being parsed and the names of the classes found are collected in the *base class set* (CS_b), which already contains `rdfs:Resource`, the superclass of all RDF user classes:

```
$CS_b$ := { rdfs:Resource }
foreach $\langle S, P, O \rangle \in RDFS$
    if $P$=rdf:type and $O$=rdfs:Class
    then $CS_b := CS_b \cup \{ S \}$
```

where *RDFS* represents the set of all subject-predicate-object triples found in the RDF Schema documents.

There also exists the *derived class set* (CS_d), containing the names of the derived classes, i.e. classes which lie at rule heads (*conclusions*). CS_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 (or a negated atom).

```
$CS_d := \varnothing$
foreach $c \in rel(\_opr(atom(\_head(imp)))) \cup rel(\_opr(atom(neg(\_head(imp)))))$
    $CS_d := CS_d \cup \{ c \}$
```

The function $f_1(f_2(\ldots f_n(x)))$ evaluates the XPath expression `//x/f`$_n$`/…/f`$_2$`/f`$_1$ and returns the corresponding node-set. When there is a single clause *f*, it simply corresponds to the expression `//f`. Attributes are retrieved via the composite function *@f*, which corresponds to the expression `//@f`. CS_d 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. Notice that in some rules the `atom` element may not be the direct child of the `_head` element because a `neg` element may lie in between.

The union of the above two sets results in the *full class set* CS_f ($CS_f := CS_b \cup CS_d$), which is used for constraining the allowed class names, when editing the contents of the `rel` element inside `atom` elements of the rule body.

3.2.2. *Determining class boxes, class patterns and slot patterns*

Class Boxes, Class Patterns and Slot Patterns are objects needed to visualize the class-related nodes of the rule graph. The structure of these objects is depicted in Table 1.

Table 1. The structure of class boxes, class patterns and slot patterns.

Class name	**Attributes**	**Explanation**
Class Box (C_B)	*N*	Class box name
	P	The set of class patterns of a class box
Class Pattern (*C_P*)	*N*	Name of corresponding class box
	Body	The rule in the body of which the class pattern appears
	Head	The rule in the head of which the class pattern appears
	S	The set of slot patterns of a class pattern
	In	The rule arrow that ends in the class pattern (when the class pattern is a conclusion of a rule)
	Out	The class arrow that emanates from the class pattern (when the class pattern is in a rule body)
Slot Pattern (*S_P*)	*N*	Slot pattern name
	Var	The list of variables of a slot pattern
	Constraint	The list of constraints of a slot pattern

For each class *c,* a *class box cb* with the same name is constructed and placed inside the corresponding *class box set* CB_b, CB_d and CB_f:

```
$CB_b := \emptyset$
foreach $c \in CS_b$
    create $cb$ of class $C\_B$
    $CB_b := CB_b \cup \{ cb \}$
    $cb.N := c$
```

```
$CB_d := \emptyset$
foreach $c \in CS_d$
    create $cb$ of class $C\_B$
    $CB_d := CB_d \cup \{ cb \}$
    $cb.N := c$
```

$$CB_f := CB_b \cup CB_d$$

Class boxes are initially empty and are dynamically populated with one or more class patterns as follows: for each `atom` element *a* inside a rule head or body, a new class pattern *cp* is created and is inserted into the class box, whose name *cb* matches the class name that appears inside the specific atom. The set of all class patterns is denoted by *CP*.

```
$CP := \varnothing$
foreach $r \in imp$
    foreach $a \in atom(\_body(r)) \cup atom(neg(\_body(r)))$
        foreach $cb \in CB_f$
            $cb.P := \varnothing$
            if $cb.N = rel(\_opr(a))$
            then
                create $cp$ of class $C\_P$
                $cb.P := cb.P \cup \{ cp \}$
                $cp.N := cb.N$
                $cp.Body := @ruleID(\_rlab(r))$
                $CP := CP \cup \{ cp \}$
```

There is a corresponding procedure for the class patterns of the rule heads:

```
foreach $r \in imp$
    foreach $a \in atom(\_head(r)) \cup atom(neg(\_head(r)))$
        foreach $cb \in CB_f$
            $cb.P := \varnothing$
            if $cb.N = rel(\_opr(a))$
            then
                create $cp$ of class $C\_P$
                $cb.P := cb.P \cup \{ cp \}$
                $cp.N := cb.N$
                $cp.Head := @ruleID(\_rlab(r))$
                $CP := CP \cup \{ cp \}$
```

Similarly to class boxes, class patterns are empty, when they are initially created, but are soon populated with one or more slot patterns. For each `_slot` element inside an atom, a slot pattern *sp* is created that consists of a slot name (contained inside the corresponding attribute) and, optionally, a variable and a list of value constraints. Slot pattern *sp* is then inserted into the storage of the class pattern *cp* that corresponds to the relevant atom *a*. The set of all slot patterns is denoted by *SP*.

```
$SP := \varnothing$
foreach $\alpha \in atom$
    foreach $s \in @name(\_slot(a))$
        foreach $cb \in CB_f$
            foreach $cp \in cb.P$
                if $cb.N = rel(\_opr(a)))$
                then
                    create $sp$ of class $S\_P$
                    $SP := SP \cup \{ sp \}$
                    $sp.N := s$
                    $cp.S := cp.S \cup \{ sp \}$
```

Each of the slot pattern parts (slot name, variable and list of value constraints) is being retrieved from the children (direct and indirect) of the `_slot` element in the XML tree representation of the rule base.

```
foreach $\alpha \in atom$
    foreach $s \in \_slot(a)$
        foreach $v \in var(s) \cup var(\_and(s))$
            foreach $cb \in CB_f$
                foreach $cp \in cb.S$
                    foreach $sp \in cp.S$
                        if $cb.N=rel(\_opr(a)) \wedge sp.N=@name(s)$
                        then $sp.Var := sp.Var \cup \{ v \}$
        foreach $c \in ind(s) \cup \_not(s) \cup ind(\_and(s)) \cup function\_call(\_and(s)))$
            foreach $cb \in CB_f$
                foreach $cp \in cb.S$
                    foreach $sp \in cp.S$
                        if $cb=rel(\_opr(a)) \wedge sp.N=@name(s)$
                        then $sp.Constraint := sp.Constraint \cup \{ c \}$
```

3.2.3. *Rule circles and arrow types*

Rule Circles and Arrows are objects that are needed in order to visualize the rule nodes and the arcs of the rule graph. The structure of these objects is depicted in Table 2.

Note that some of the attributes above are applied later on, in section 3.2.4, where the algorithm for visualizing a rule base is thoroughly presented.

Table 2. The structure of rule circles and arrows.

Class name	**Attributes**	**Explanation**
Rule Circle (R_C)	*N*	Rule name
	In	The set of incoming arrows
	Out	The outgoing arrow
Rule Arrow (*R_A*)	*N*	Rule name
	In	The rule circle from which the arrow emanates
	Out	The class box node to which the arrow ends
	Type	The arrow type (plain|expandable – see section 3.2.4)
	Orient	The arrow orientation (plain|dotted – see section 3.2.4)
Superiority Arrow (*SR_A*)	*SUP*	The superior rule of the superiority relation
	INF	The inferior rule of the superiority relation
	In	The rule circle from which the arrow emanates
	Out	The rule circle to which the arrow ends
Class Arrow (*C_A*)	*In*	The class pattern node from which the arrow emanates
	Out	The rule circle to which the arrow ends
	N	A tuple of the class pattern and the corresponding rule that uniquely identifies the class arrow

For every rule in the rule base a rule circle is constructed, whose name matches the value of the `ruleID` attribute in the `_rlab` element of the corresponding rule. The set of all rule circles is denoted by *RC* and all rules are included in the rule set *RS*. The rule type is equal to the value of the `ruletype` attribute inside the `_rlab` element of the respective rule and can only take three distinct values (`strictrule`, `defeasiblerule`, `defeater`). The corresponding arrow sets are denoted by *SA*, *DA* and *FA*. The set of all arrows originating from rule circles is denoted by *RA*. Rule circles are connected with the arrows representing rules, regardless their type.

```
$RS := RC := SA := DA := FA := \varnothing$
foreach $r \in imp$
    $RS := RS \cup \{ r \}$
    create $rc$ of class $R\_C$
    create $ar$ of class $A\_R$
    $rc.N := ar.N := @ruleID(\_rlab(r))$
    $RC := RC \cup \{ rc \}$
    $rc.Out := ar$
    $ar.In := rc$
    if $@ruletype(\_rlab(r)) = strictrule$
    then
        $SA := SA \cup \{ ar \}$
        $rc.Type := strictrule$
    if $@ruletype(\_rlab(r)) = defeasiblerule$
    then
        $DA := DA \cup \{ ar \}$
        $rc.Type := defeasiblerule$
    if $@ruletype(\_rlab(r)) = defeater$
    then
        $FA := FA \cup \{ ar \}$
        $rc.Type := defeater$
$RA = SA \cup DA \cup FA$
```

The superiority relationship is represented as an attribute (`superior`) inside the superior rule element. For each such relationship, a superiority arrow object is created, linking the superior rule with the inferior rule. The set of all superiority arrows is *SRA*.

```
$SRA := \varnothing$
foreach $r \in imp$
    foreach $sr \in @superior(\_rlab(imp))$
        create $sar$ of class $SR\_A$
        $SRA := SRA \cup \{ sar \}$
        $sar.SUP := sar.In := @ruleID(\_rlab(r))$
        $sar.INF := sar.Out := sr$
        $r.Out := sr.In := sar$
```

The arrows between the class patterns of the rule body and the rule circles are contained in the *CA* set:

```
$CA := \emptyset$
foreach $cp \in CP$
    create $car$ of class $C\_A$
    $CA := CA \cup \{ car \}$
    $car.N := \langle cp, cp.Body \rangle$
    $cp.Out := cp.Body.In := car$
    $car.In := cp$
    $car.Out := cp.Body$
    $car.Out.Premises := car.Out.Premises \cup \{ cp \}$
```

where the $\langle cp, cp.Body \rangle$ tuple uniquely identifies such arrows, because the same class pattern can be re-used in the body of many rules.

What remains to be established is how the arrows between the rule circles and the class patterns of the rule head are constructed. These arrows are contained in the *RA* set, presented above. Class patterns of the rule head are connected to rule arrows as follows:

```
foreach $ar \in RA$
   foreach $cp \in CP$
      if $cp.Head = ar.N$
      then
         $cp.In := ar$
         $ar.Out := cp$
         $ar.In.Conclusion := cp$
```

3.2.4. *The visualization algorithm*

After having collected all the necessary graph elements and having populated all the class boxes with the appropriate class and slot patterns, three sets exist: (i) the base class boxes set CB_b that contains the class boxes corresponding to base classes, (ii) the derived class boxes set CB_d that contains the class boxes corresponding to derived classes, and (iii) the set *RC* that includes all the rule circles of the rule base.

The next important task is the placement of each element in the graph. To this end, an algorithm for the visualization of the rule base was implemented, which utilizes common rule stratification techniques.[14] Unlike the latter, however, that focus on computing the minimal model of a rule set, our algorithm aims at the optimal *visualization* outcome, namely the simplest graph possible. The algorithm is displayed in Fig. 8.

The algorithm gives a left-to-right orientation to the flow of information, placing the graph elements in strata (or columns), with the first stratum located on the utmost left and the numbering of the strata following the same left-to-right orientation. In other words, the proposed algorithm deals with the "*stratification*" of the graph elements, calculating the optimal stratum, in which each graph element has to be placed.

During the execution of the algorithm, the following steps can be distinguished:

(i) All the base class boxes are placed in stratum #1.
(ii) The algorithm enters a loop, consecutively assigning strata to rule circles and derived class boxes, incrementing each time the stratum counter by 1.
  (a) A rule circle is assigned to a stratum, when all its premises belong to previous strata, with at least one of them belonging to the immediately previous stratum.
  (b) A class box is assigned to a stratum, if it contains the conclusions of rules in the immediately previous stratum.

```
str:=1
foreach cb∈CB_b do cb.Stratum:=str
while |RC|≠0 do
   RuleTemp:=∅
   str:=str+1
   foreach R∈RC do
      if ((∀p∈R.Premises → p.N.Stratum<str) ∧
            (∃p'∈R.Premises ∧ p'.N.Stratum)=str-1))
      then R.Stratum:=str, RC:=RC-{R}, RuleTemp:=RuleTemp∪{R}
   foreach R∈RuleTemp do
      foreach p∈R.Premises do
            if p.N.Stratum=str-1
            then Type:=plain else Type:=expandable,
            foreach X∈R.In do
                  if X.In=p ∧ X.Out=R
                  then X.Type := Type
   str:=str+1
   CbTemp:=∅
   foreach R∈RuleTemp do
      if unknown(R.Conclusion.N.Stratum)
      then R.Conclusion.N.Stratum:=str,
      CbTemp:=CbTemp∪{R.Conclusion.N}
   foreach R∈RuleTemp do
      if R.Conclusion.N∈CbTemp
      then Orient:=plain else Orient:=dotted,
      foreach X∈R.Out do
            if X.In=R ∧ X.Out=R.Conclusion
            then X.Orient:= Orient
```

Fig. 8. The rule stratification algorithm.

In the cases of cycles in the graph (i.e. a conclusion of a rule serves as a premise for another rule in a previous stratum), neither the conclusion is drawn again, nor the arrow connecting the rule with the conclusion is drawn backwards. Instead, in order to prevent graph cluttering, a special type of "*dotted*" arrow is applied, commencing from the rule circle and ending in three dots "**...**". By clicking on the arrow, the user is presented with a popup window, displaying the rule at full detail, including its premises and conclusion.

Also, according to the algorithm, only the arcs that connect two *consecutive* graph elements are drawn by default. When the stratum difference between a class pattern and a rule circle is greater than 1, the arrow that connects them is qualified as "*expandable*" (contrary to "*plain*"). Expandable arrows are not drawn by default, but can be included in the graph, by "*expanding*" (or *revealing*) all the arcs of the corresponding rule.

3.2.5. *Example*

This section outlines an example that better illustrates the representation approach as well as the functionality of the algorithm described in the previous sections. Suppose that we have the following rule base:

r_1: `hard_rock(X)` $\rightarrow$ `rock(X)`
r_2: `rock(X)` $\Rightarrow$ `likes(X)`
r_3: `hard_rock(X)` $\Rightarrow$ `¬likes(X)`
r_4: `hard_rock(X), artist(X,"Rainbow")` $\Rightarrow$ `likes(X)`

The first three rules were encountered in section 2.1, while rule r_4 reads as "*Alice likes hard rock songs by Rainbow*".

In VDR-DEVICE the OO data model is used (instead of the predicate/relational model). So, predicates like `rare` and `artist` are actually represented as attributes of the class `hard_rock`. Therefore, in the example three classes are needed, as Table 3 indicates: one base class (`hard_rock`) and two derived classes (`rock` and `likes`). Thus, the three "key" sets, as described in sections 3.2.2 and 3.2.3 will be formulated as follows:

CB_b := { `hard_rock` }
CB_d := { `rock`, `likes` }
RC := { r_1, r_2, r_3, r_4 }

After applying the algorithm, it comes up that four strata are needed to display all the graph elements. Table 4 displays the final stratum assignments, according to the algorithm. The first stratum is mapped to the first column on the left, the second stratum to the column on the right of the first one and so on. Nodes in one column are never connected with nodes in the same column, except from the case of rule superiority.

Table 3. Classes, included in the example.

Base Class	`hard_rock`
Derived Classes	`rock, likes`

Table 4. Stratum assignments.

stratum #1	`hard_rock`
stratum #2	r_1, r_3, r_4
stratum #3	`rock, likes`
stratum #4	r_2

Figure 9 displays the resulting graph, produced by DR-VisMo, which is compliant with the algorithm and the representation approach presented previously. The main window of the program is composed of a left-hand-side panel, where the rule graph is displayed and a right-hand-side panel, which shows the properties of the graph element selected on the left. Notice the "dotted" arrow "leaving" rule r_2. As explained earlier, this arrow type is applied in cases of rule conclusions appearing in earlier strata than the rule. By clicking on the arrow, a pop-up window presents the user with details, regarding the corresponding rule, displaying its premises and conclusion.

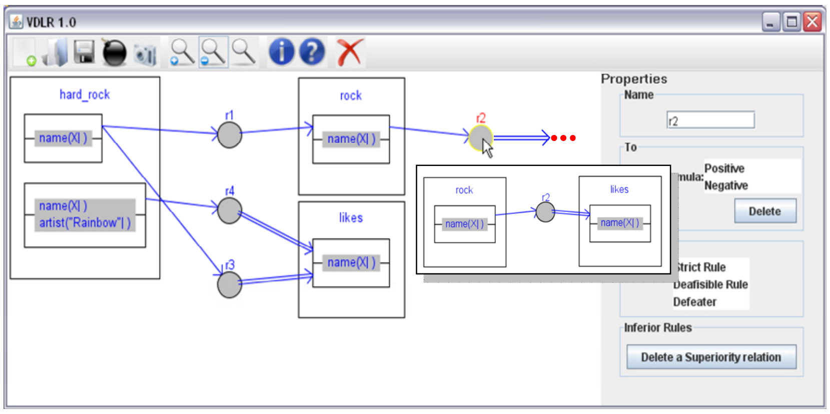


Fig. 9. Implementation of the visualization algorithm.

3.3. *System architecture and functionality*

The previous sections focused on the various elements of the rule graph, while the following subsections describe in detail the five modules that comprise DR-VisMo. The overall architecture of the system is displayed in Fig. 10.

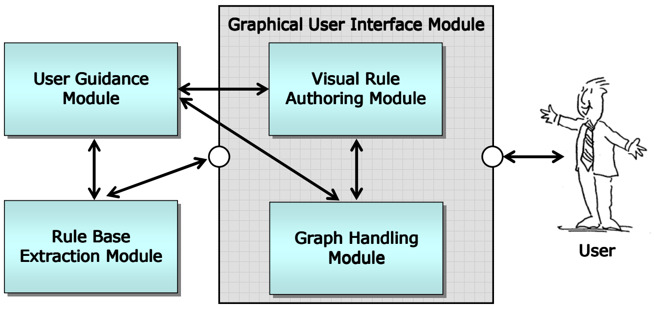

Graphical User Interface Module
User Guidance Module
Visual Rule Authoring Module
Rule Base Extraction Module
Graph Handling Module
User

Fig. 10. DR-VisMo architecture.

3.3.1. *Visual rule base authoring module*

This module is the backbone of the system, since its primary function deals with the visual development and visualization of the rule base created. More specifically, the module is responsible for the creation of the visual constructs (rule circles, class boxes, class patterns, slot patterns) that represent rules and atomic formulas in the graph as well as the connections among the various graph elements (see section 3.1). Furthermore, the rule base authoring module is also responsible for rendering the rule graph, not only throughout its development stages, but also during the points, when users browse certain parts of the graph or when they move around the graph or its various subparts.

The module communicates closely with two other modules: the graph handling and the user guidance module, described in the following two subsections. The graph handling module is responsible for displaying the corresponding properties of the graph elements. Therefore, since the two modules present different aspects of the same rule base, there cannot be any inconsistencies in the information they show. As for the latter module, it is triggered by the authoring module during the development of a rule base for controlling the input data and preventing syntactic and semantic errors, by displaying relevant error messages.

3.3.2. *Graph handling module*

This module is primarily responsible for handling the information, stored in the rule graph. More specifically, it displays the properties, corresponding to the various graph elements and is responsible for handling changes, performed by the user. The following modifications are allowed:

- Changing the name of visual constructs.
- Appending class patterns into class boxes.
- Inserting slot patterns into or removing slot patterns from a class pattern.
- Handling the connections among the visual constructs (positive/negated atomic formula, rule type, superiority relationships).
- Removing connections among visual constructs.

These modifications are automatically reflected to the graph visualization, since the graph handling module closely cooperates with the rule base authoring module. Furthermore, this module also cooperates with the user guidance module, for preventing erroneous or inconsistent alterations to the properties of the graph elements.

3.3.3. *User guidance module*

The user guidance module accepts input from the previous two modules and is responsible for detecting and correcting potential errors on behalf of the user. Its main functionality deals with inspecting the created slot patterns for syntactic validity (see section 3.1 and Fig. 7). In case of a syntactic error, the module has to detect the mistake, point the user to it and propose ways of correcting it, by displaying relevant error messages.

3.3.4. *User interface module*

The user interface module comprises the sole means of interaction with the user. Therefore, its design rationale includes user-friendliness and efficiency. Furthermore, besides containing the rule authoring and graph handling modules, the user interface is also responsible for communicating with the RuleML extraction module that will encode the defeasible logic rule base in the RuleML-compatible syntax of DR-VisMo.

3.3.5. *Rule base extraction module*

This module is primarily concerned with encoding the developed rule graph into a RuleML-compatible document. This, however, is a two-step process: firstly, the module receives the graph information from the user interface and creates an intermediary file that corresponds to the rule base. Then, an XSLT transformation is performed, which transforms the latter file into a RuleML-compatible document. Depending on the desired RuleML version of the exported file, a different XSLT transformation is applied. Currently, the system supports RuleML versions 0.86 and 0.91, but can be easily extended to support more recent (or older) versions as well.

4. Evaluation

DR-VisMo was evaluated by post-graduate students (25 in total) attending a Semantic Web course at the Department of Informatics at our university. The students were given a defeasible logic rule base in textual form and were asked to model it using the software. They were also requested to answer an on-line questionnaire for assessing DR-VisMo. The questionnaire was not aimed at the usability of the software facilities; instead, its primary objective was to allow the users to evaluate the representation schema adopted by the system. The survey was divided in two major parts: the first part contained questions, related to the intuitiveness and user-friendliness of the proposed representation of defeasible logic rule bases, while the second part asked the users to evaluate the degree of assistance that this representation schema offers during the development of a rule base.

Regarding the former part of the survey, users generally seemed to understand and appreciate the adopted representation methodology. More specifically, 72% of the participants found the representation intuitive, 88% found it easy to understand, 80% found it aesthetically satisfactory, all of them (100%) found it interesting, while only 12% found it incomprehensible and unacceptable.

The evaluation results, regarding the latter part were also encouraging: all of the participants (100%) believed that DR-VisMo indeed assists in the development of a defeasible logic rule base, 76% believed that the system considerably improves productivity (i.e. minimizes development time), 72% considered that the representation gives a better overview of the rule dependencies, while only 16% of the users would rather use another tool.

Overall, the result of using the system was considered acceptable and impressive by 44% and 30% of the users, respectively. There were users, nevertheless, that would prefer more features in the proposed representation (32%), while, on the other hand, DR-VisMo made defeasible logic attractive to 76% of the participants. Some shortcomings that users detected and will be dealt with in our future improvements of the system include handling more than one variable in a class pattern, representing conflicting literals and including negation-as-failure in the representation.

5. Related Work

d-GRAPHER[15] is system that consists of a visual defeasible graph (d-graph) editor and a prolog-based inference engine. The system includes error-checking routines that prevent the construction of illegal graphs, displaying appropriate error messages. Although d-GRAPHER was the first system that offered visual development of d-graphs, adopting a representation that comprised the starting point for DR-VisMo, it presents, nevertheless, a number of drawbacks: the rule bases produced are of an elementary level of expressiveness, not allowing conjunction/disjunction of atoms or representation of slot variables and value constraints. Furthermore, the system is not able to represent more expressive rule bases and is, thus, an isolated solution.

To the best of our knowledge, no other visual defeasible reasoning rule editors exist. There exist, however, several visual editors for other types of logic. The editor of *TIGER*[16] is such an example. TIGER is an environment for modeling domains, using a visual language. One of its subsystems is its visual rule editor, which deals with graph rules. In TIGER, graph rules are used to manipulate the graph representation of a language element and define syntax-directed editor commands. Since TIGER and VDR-DEVICE are two highly differentiated systems, it is pointless to compare the respective visual rule editors. Nevertheless, according to its designers, TIGER has been successfully applied in a variety of scenarios, including the design of activity and sequence diagrams, Petri nets and automata.

On the other hand, several graphical rule editors exist. The *Protégé SWRL Editor*,[17] for instance, is such a case. As its name implies, the software operates within *Protégé-OWL*[18] and allows users to seamlessly switch between SWRL rule editing and editing of OWL entities, also allowing for incorporation of OWL entities into rules. SWRL Editor deals with Horn-like rules, expressed in terms of OWL concepts, offering, however, a lower degree of expressiveness, since it does not support the development of defeasible logic rules. Another difference with DR-VisMo is the fact that the former is graphical, contrary to the latter, which is purely visual.

Another SWRL editor is *RuleVisor*,[19] implemented as part of the SAWA (Situation Awareness Assistant) framework. The editor assists in the construction and maintenance of SWRL rules, also cooperating with ontologies that provide the content, upon which a rule set is to be built. The tool offers a user-friendly, yet frame-based and elementary, development environment, which, especially in the cases of rule bases of a considerable size, proves to be rather impractical.

A further paradigm of a graphical rule editor is *Oryx*, the graphical front-end for *Mandarax*,[20] which is a pure object oriented rule base engine for deduction rules. Oryx consists of two parts, namely, a standalone and a server application. Oryx supports verbalization of knowledge and, thus, includes a formal natural-language-based rule editor as well as a repository for managing the vocabulary and a graphical user interface library. Nevertheless, similarly to other aforementioned systems, Oryx cannot model defeasible logic rules and it also comprises a graphically-based rule editor paradigm.

On the other hand, there exists a variety of systems that implement rule representation and visualization, although we didn't come across any system that can visually represent defeasible logic rules yet. Such an example is *VisiRule*,[21] a graphical tool (module of LPA Win Prolog) for delivering business rule and decision support applications. The user draws a flowchart that represents the decision logic and VisiRule produces Flex code and compiles it. The system offers guidance during the construction process, constraining errors, based on the semantic content of the emerging program. This reduces the possibility of constructing invalid or meaningless links, improving productivity and helping detect errors early within the design process.

Finally, a knowledge representation paradigm, "borrowed" from the domain of Law, is *eGanges*,[22] a legal expert system shell that provides negotiation support for domestic conflict prevention applications. The system employs a plug-in, called *Negaid* that adopts a knowledge representation scheme, which is a tree-like structure, called a "river". The nodes in the river are antecedents in the conditional propositions of the procedural or conflict resolution rule system, while arrows represent inference and indicate the flow of extended deduction. Although eGanges and DR-VisMo both display inference steps and interrelations among rules and antecedents, the two systems are remotely associated and deal with differentiated domains; thus a comparison here regarding their functionalities would be rather misplaced.

6. Conclusions and Future Work

This paper argued that logic and rules are the primary means of realizing the Semantic Web vision, by offering the ability of reasoning over the information scattered in the Web. Defeasible reasoning was proposed as an approach that can assist towards this affair. It represents a non-monotonic reasoning approach that features high expressiveness and low computational overhead. Furthermore, it is highly suitable for the Semantic Web, since it can handle inconsistent and conflicting information, a situation that is often encountered in highly dynamic environments like the Web.

Defeasible reasoning is based on a solid mathematical notation, which sometimes may seem confusing to the end user. A system, called DR-VisMo, for authoring and visualizing defeasible logic rule bases was presented in this paper, aiming at alleviating this problem. DR-VisMo adopts a representation schema, based on enhanced directed graphs that feature distinct node types for rules and atomic formulas and distinct connection types for each rule type in defeasible logic and for superiority relationships. For the visualization of defeasible logic rule bases, DR-VisMo also implements a stratification algorithm that deals with decisions, regarding the arrangement of the various elements in the graph.

The motivation behind DR-VisMo is associated with the lack of tools for visual modeling and representation of rule bases in general and defeasible logic rule bases in particular. As seen in the "Related Work" section, no modern systems exist currently for authoring defeasible reasoning rule bases, while there is also a deficiency of tools for rule bases of other types of logics. And, as the user evaluation presented earlier (see section 4) demonstrates, the system indeed seems to assist users in the development of defeasible logic rule bases, significantly reducing the time needed for the task. The assessment results were overall encouraging.

As for our future research goals, a variety of tasks still remains to be addressed. Potential improvements of DR-VisMo include enhancing visual representation with negation-as-failure and variable unification, for simplifying the display of multiple unifiable class patterns. The improvement of the syntax control module is also necessary, for offering better guidance to the end user, during the addition of new slots and conditions inside a class pattern. Other improvements were also proposed by the users, who participated in the evaluation.

Expressive visualization of a defeasible logic rule base can lead to proof explanations. By adding visual rule execution tracing, proof visualization and validation to DR-VisMo, we will then be able to delve deeper into the Proof layer of the Semantic Web architecture, implementing facilities that would increase the trust of users towards the Semantic Web.

Acknowledgments

This work was partially supported by a NON-EUROPE project (GSRT - 05 NON EU 423), jointly funded by the European Union and the Greek Government (General Secretariat of Research and Technology/GSRT).

References

1. T. Berners-Lee, J. Hendler and O. Lassila, The Semantic Web, *Scientific American*, **284**(5), 2001, pp. 34-43.
2. D. Nute, Defeasible Reasoning, *Proc. 20th Int. Conf. on Systems Science*, IEEE Press, 1987, pp. 470-477.
3. M. A. Covington, D. Nute and A. Vellino, *Prolog Programming in Depth*, Prentice-Hall, Inc. 1996.
4. M. J. Maher, Propositional Defeasible Logic has Linear Complexity, *Theory and Practice of Logic Programming*, **1**(6), 2001, pp. 691–711.
5. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1994.
6. E. Kontopoulos, N. Bassiliades and G. Antoniou, Visualizing Defeasible Logic Rules for the Semantic Web, *Proc. 1st Asian Semantic Web Conf.* (*ASWC'06*), Springer-Verlag, LNCS 4185, Beijing, China, 2006, pp. 278-292.
7. E. Kontopoulos, N. Bassiliades and G. Antoniou, Visual Stratification of Defeasible Logic Rule Bases, *Proc. 19th IEEE Int. Conf. on Tools with Artificial Intelligence* (*ICTAI*), IEEE, Patras, Greece, 2007, pp. 238-245.
8. N. Bassiliades, E. Kontopoulos and G. Antoniou, A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web, *Proc. RuleML-2005*, Galway, Ireland, Springer-Verlag, LNCS 3791, 2005, pp. 172-186.
9. D. Nute and K. Erk, Defeasible logic graphs: I. Theory, *Decis. Support Syst.*, **22**(3), 1998, pp. 277-293.
10. Antoniou, G., Billington, D., Governatori, G., Maher, M. J. Representation results for defeasible logic. *ACM Trans. Comput. Log.*, **2**(2), 2001, pp. 255-287.

11. N. Bassiliades, G. Antoniou and I. Vlahavas, A Defeasible Logic Reasoner for the Semantic Web, *Int. Journal on Semantic Web and Information Systems*, **2**(1), 2006, pp. 1-41.
12. N. Bassiliades and I. Vlahavas, R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata, *Int. Journal on Semantic Web and Information Systems*, **2**(2), 2006, pp. 24-90.
13. CLIPS Basic Programming Guide (v. 6.24), www.ghg.net/clips/CLIPS.html, last accessed: January 3, 2008.
14. J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, 1988.
15. D. Nute, Z. Hunter and C. Henderson, Defeasible logic graphs: II. Implementation, *Decis. Support Syst.*, **22**(3), 1998, pp. 295-306.
16. C. Ermel, K. Ehrig, G. Taentzer and E. Weiss, Object Oriented and Rule-based Design of Visual Languages using TIGER, *Proc. Third Int. Workshop on Graph-Based Tools (GraBaTs'06)*, volume 1, Natal, Brazil, September 2006, Electronic Communications of the EASST.
17. M. J. O'Connor, H. Knublauch, S. W. Tu, B. Grossof, M. Dean, W. E. Grosso and M. A. Musen, Supporting Rule System Interoperability on the Semantic Web with SWRL, *Proc. 4th International Semantic Web Conference* (*ISWC2005*), Galway, Ireland, 2005.
18. H. Knublauch, R. W. Fergerson, N. F. Noy and M. A. Musen, The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications, *Proc. of the 3rd Int'l Semantic Web Conf. ISWC*), Hiroshima, Japan, 2004, pp. 229-243.
19. C. Matheus, M. Kokar, K. Baclawski and J. Letkowski, An Application of Semantic Web Technologies to Situation Awareness, *Proc. 4th International Semantic Web Conference* (*ISWC 2005*), Galway, Ireland, 2005.
20. J. Dietrich, A. Kozlenkov, M. Schroeder and G. Wagner, Rule-based agents for the semantic web, *Electronic Commerce Research and Applications*, **2**(4), 2003, pp. 323–338.
21. R. Shalfield, VisiRule User Guide, http://www.lpa.co.uk/ftp/4600/vsr_ref.pdf, 2005.
22. P. N. Gray, X. Gray and J. Zeleznikow, Negotiating logic: for richer or poorer, *Proc. 11th international Conference on Artificial intelligence and Law* (*ICAIL*), Stanford, California, 2007, ACM, New York, NY, 247-251, DOI= http://doi.acm.org/10.1145/1276318.1276366.