Visual Stratification of Defeasible Logic Rule Bases

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

Logic and proofs constitute key factors in increasing the user trust towards the Semantic Web. Defeasible reasoning is a useful tool towards the development of the Logic layer of the Semantic Web architecture. However, having a solid mathematical notation, it may be confusing to end users, who often need graphical trace and explanation mechanisms for the derived conclusions. In a previous work of ours, we outlined a methodology for representing defeasible logic rules, utilizing directed graphs that feature distinct node and connection types. However, visualizing a defeasible logic rule base also involves the placement of the multiple graph elements in an intuitive way, a non-trivial task that aims at improving user comprehensibility. This paper presents a stratification algorithm for visualizing defeasible logic rule bases that query and reason about RDF data as well as a tool that applies this algorithm.

1. Introduction

The mature steps towards accomplishing the Semantic Web vision have reached as far as the development of ontologies and OWL [13], the Web Ontology Language, which is now the dominant standard in ontology encoding. The upcoming efforts will be targeted at logic and proofs, which are believed to possess a key role in assisting users towards eventually accepting the Semantic Web.

Defeasible reasoning [8], a member of the non-monotonic reasoning family, represents a rule-based approach to reasoning with incomplete and conflicting information. It can represent facts, rules, priorities and conflicts among rules. Nevertheless, defeasible reasoning features a solid mathematical notation, which may seem confusing to end users, who often need graphical trace and explanation mechanisms for the derived conclusions.

Directed graphs (or *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.

The basic theoretical principles for representing defeasible logic rules using digraphs were presented in a previous work of ours [7]. 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. However, visualizing an entire rule base involves decisions regarding the arrangement of the various graph elements, a task that considerably improves clarity.

This paper presents a stratification algorithm for visualizing defeasible logic rule bases as well as a software tool that applies this algorithm. The tool, called *dl-RuleViz*, can represent defeasible logic rule bases that query and reason about RDF data, taking into account the semantics of RDF Schema ontologies. The software is implemented as part of *VDR-DEVICE* [2], an environment for modeling and deploying defeasible logic rule bases on top of RDF ontologies. Notice that stratification is solely used for visualization purposes and is indifferent for 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 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 illustrates the principles of visualizing a defeasible logic rule base, focusing on the proposed stratification algorithm and dl-RuleViz, while section 4 discusses related work, followed by the conclusions and directions for future research.

2. Digraphs and Defeasible Logics

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 substantially based on the methodology presented by Nute in [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, the digraphs in our approach contain two kinds of nodes: (a) literals, represented by rectangles, called "*literal boxes*" and (b) rules, represented by circles. Furthermore, in defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters; each one of the rule types is mapped to one of three distinct connection types, similarly to [9]. This results in rules of different types being represented more distinctively.

2.1. Rule Types in Defeasible Logic

The full theoretical approach, regarding the graphical representation of defeasible reasoning elements is thoroughly described in [7]; here only a brief outline will be made. Thus, the first rule type in defeasible reasoning is *strict rules*, which are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, then so is the conclusion. An example of a strict rule is: "*Novels are books*", which formally would become: r_1: novel(X) → book(X), and would be represented by the digraph in Fig. 1.

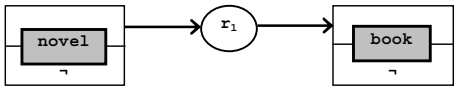


Fig. 1. Visual representation of strict rule r_1

Each literal box consists of two adjacent "*atomic formula boxes*", where the upper one represents a positive and the lower one a negated atomic formula. This way, atomic formulas are depicted together distinctively, maintaining their independence. Notice also that in the rule graph we only represent the predicate and not the literal (i.e. predicate plus all the arguments), because we are currently interested in clarifying the interrelationships between the concepts (through the rules) and not the complete defeasible theory details. Nevertheless, the final representation (presented later) implements 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: book(X) ⇒ hardcover(X) ("*Books are typically hard-covered*") and r_3: novel(X) ⇒ ¬hardcover(X) ("*Novels are typically not hard-covered*"). Both are depicted in Fig. 2.

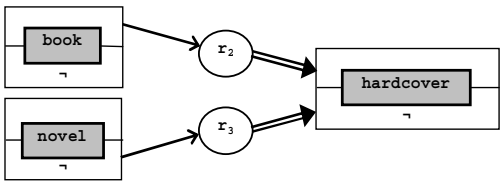


Fig. 2. Representing defeasible rules r_2, r_3

Defeaters, denoted by $A \sim\!> p$, do not actively support conclusions, but can only prevent some of them. An example is: r_2': cheap(X) ~> ¬hardcover(X) that reads as: "*Cheap books are not hard-covered*". This defeater can defeat, for example, rule r_2 mentioned above and it can be represented by Fig. 3.

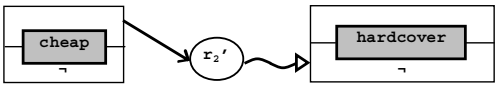


Fig. 3. Visual representation of defeater r_2'

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 novels are eventually hard-covered 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 novels are not hard-covered. In this case, rule r_3 is called *superior* to r_2 and r_2 *inferior* to r_3. In the case of superiority relationships a fourth connection type is introduced, displayed in Fig. 4.

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

2.2. Representing Arguments and Conditions

So far we have demonstrated 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 X > 4, 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 a(X,2) 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 fragment `a(X,Y),Y>4` appears in a rule condition, it can be represented as in Fig. 5 (b).

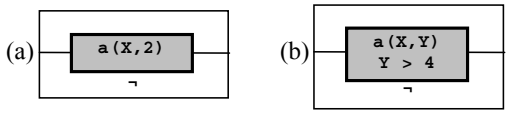


Fig. 5. Representing (a) arguments of literals and (b) simple conditions on variables

A certain predicate, say `a`, 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 "lose" the predicate name, which is now located at the top of the predicate box. The literal boxes inside predicate boxes that express conditions on instances of the specific predicate extension are called *predicate patterns*.

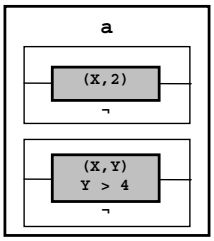


Fig. 6. Predicate box and predicate patterns

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.

3. Visualizing a Defeasible Logic Rule Base

VDR-DEVICE (Visual DR-DEVICE) [2] is a visual, integrated development environment for modeling and deploying defeasible logic rule bases on top of RDF ontologies. It consists of two primary components: (i) *DR-DEVICE*, the reasoning system that processes RDF data, performs the defeasible inference procedure, produces the results and exports them as RDF data; (ii) *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.

Moreover, DRREd is equipped with *dl-RuleViz*, a module that allows users to visualize defeasible logic rule bases. Besides the XML-tree format, featured by DRREd, dl-RuleViz offers an additional graph-based means of visualizing the rule base. The following subsections present key-aspects of this representation.

3.1. Brief Description of the Reasoning System

DR-DEVICE 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 [3].

DR-DEVICE rule bases are expressed in an extension of RuleML (see [1] for a reference and DTD). Extensions deal with two aspects of DR-DEVICE, namely defeasible logic and its CLIPS [4] 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 [1]. RDF class names, used as base classes in the rule condition, are referred to via the `href` attribute of the `rel` element (e.g. `novel` in Fig. 7), 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 ([1], [3]).

3.2. Collecting the Class Names

The RDF Schema documents, designated by the DRREd 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. Therefore, the CS_b set is constructed as follows:

`rdfs:Resource` $\in CS_b$

$\forall c$ (*c* `rdf:type rdfs:Class`) $\rightarrow c \in CS_b$

where (*X Y Z*) represents an RDF triple 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). This set 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.

$(\forall r \in \{<imp>\})$

$(\forall c \in rel(\_opr(atom\{\_head(r)\}))) \rightarrow c \in CS_d$

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.

3.3. Elements of the Rule Graph

The graph consists of nodes and edges. This section shows how we identify which are the nodes and connections to be displayed, according to their type.

3.3.1. Class Boxes, Class Patterns, Slot Patterns. For each class *c* that belongs to the base, derived and full class sets respectively, 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:

$(\forall c)\ c \in CS_b \rightarrow (\exists\ cb)\ cb \in CB_b$

$(\forall c)\ c \in CS_d \rightarrow (\exists\ cb)\ cb \in CB_d$

$CB_f = CB_b \cup CB_d$

Class boxes are simply containers and are the equivalent of predicate boxes, described previously. They are initially empty and are dynamically populated with one or more *class patterns*, the equivalent of predicate patterns, also referred to in a previous section. In practice, class patterns express conditions on instances of the specific class. Class boxes are populated 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 `rel` element of the specific atom. The set of all class patterns is denoted by *CP*. In the meantime, the class pattern is associated with the rule it appears and its position in the rule (head or body) is noted.

$(\forall r \in \{<imp>\})$

$(\forall a \in atom\{\_body(r)\})$

$((\forall cb \in CB_f), cb = \_opr(rel(a)))$

$\rightarrow ((\exists cp)\ cp \in CP \wedge N(cp) = cb \wedge Body(cp) = r \wedge cp \in S(cb))$

The expression $S(cb)$ represents a set attribute of object *cb*, namely the storage of class patterns for each class box, the expression $N(cp)$ represents a string attribute of *cp* that holds its corresponding class name, while the $Body(cp)$ expression denotes the rule in the body of which the class pattern appears.

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

$(\forall r \in \{<imp>\})$

$(\forall a \in atom\{\_head(r)\})$

$((\forall cb \in CB_f)\ cb = \_opr(rel(a)))$

$\rightarrow ((\exists cp \in CP)\ N(cp) = cb \wedge Head(cp) = r \wedge cp \in S(cb))$

Visually, class patterns appear as literal boxes, which were described in section 2.1. The mapping of class patterns to literal boxes 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), as stated in section 3.1. Thus, the truth value associated with each returned class instance will be either positive or negative.

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*.

$(\forall a \in \{<atom>\})$

$(\forall s \in name(\_slot(a)))$

$(\forall cb \in CB_f)$

$((\forall cp \in S(cb)), cb = rel(\_opr(a)))$

$\rightarrow ((\exists sp \in SP)\ N(sp) = s \wedge sp \in S(cp))$

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.

$(\forall a \in \{<atom>\})\ (\forall s \in \_slot(a))\ (\forall v \in var(s) \cup var(\_and(s)))$

$(\forall cb \in CB_f)\ (\forall cp \in S(cb))$

$((\forall sp \in S(cp)), cb = rel(\_opr(a)) \wedge N(sp) = name(s) \rightarrow v \in Var(sp))$

$(\forall a \in \{<atom>\})\ (\forall s \in \_slot(a))$

$(\forall c \in ind(s) \cup \_not(s) \cup ind(\_and(s)) \cup function\_call(\_and(s)))$

$(\forall cb \in CB_f)\ (\forall cp \in S(cb))$

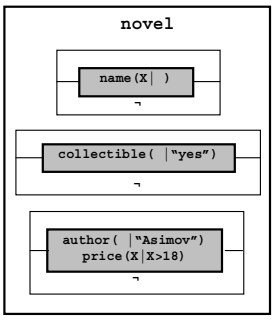
$((\forall sp \in S(cp)), cb = rel(\_opr(a)) \wedge N(sp) = name(s) \rightarrow c \in Constraint(sp))$

In the above expressions, $Var(sp)$ and $Constraint(sp)$ are the attributes that store the variable and the list of constraints of a slot pattern, respectively. However, the above expressions have been simplified for clarity reasons, since variables can be either direct children of `_slot` elements in the RuleML format, or they can be indirect descendants when combined with a constraint. Also, there is no such thing as a *constraint*

element, as suggested by the expressions above. In this case, assume that the expression *constraint*(*s*) is a function that delivers constraints from the RuleML document through a complicated, case-based algorithm. The 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).

Slot patterns 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 we have 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 not 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 that contains three class patterns applied on the *novel* class and a code fragment matching the third class pattern, written in the RuleML-like syntax of VDR-DEVICE. The first two class patterns contain one slot pattern each, while the third one contains two slot patterns. As can be observed, the argument list of each slot pattern is divided in two parts, separated by "|"; on the left all the variables are placed and on the right all the expressions and conditions, corresponding to the variables on the left. In the case of constant values, only the right-hand side is utilized; see, for instance, the second class pattern of the box in Fig. 7. Finally, the third class pattern refers to all the novels by Asimov with price greater than 18.




```
<atom>
  <_opr>
    <rel href="novel"/>
  </_opr>
  <_slot name="author">
    <ind>"Asimov"</ind>
  </_slot>
  <_slot name="price">
    <_and>
      <var>x</var>
      <function_call name="&gt;">
        <var>x</var>
        <ind>18</ind>
      </function_call>
    </_and>
  </_slot>
</atom>
```

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

3.3.2. Rule Circles and Arrow Types. Besides class boxes and their "ingredients" (class patterns, slot patterns), there also exist *circles* that represent rules and *arcs* that connect the nodes in the graph. The visual representation of rules in the digraph, using circles, was described in a previous section. There exist five types of connections in the graph: three for the rule type, one for the superiority relationship, plus a simple arrow connection type for connecting the class patterns of rule bodies to the rule circles.

For every rule r in the rule base a rule circle rc is constructed, whose name $N(rc)$ matches the name of the rule, namely the value of the `ruleID` attribute in the `_rlab` element of the corresponding rule. The set of all rule circles is denoted by RC.

$(\forall r \in \{<imp>\})\ (\exists rc \in RC)\ N(rc) = ruleID(\_rlab(r))$

All rules are included in the rule set RS:

$(\forall r \in \{<imp>\})\ (\exists r')\ r' = ruleID(\_rlab(r)) \wedge r' \in RS$

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.

$(\forall r \in \{<imp>\})\ ruletype(\_rlab(r)) = strictrule$
$\rightarrow (\exists ar \in SA)\ N(ar) = ruleID(\_rlab(r))$

$(\forall r \in \{<imp>\})\ ruletype(\_rlab(r)) = defeasiblerule$
$\rightarrow (\exists ar \in DA)\ N(ar) = ruleID(\_rlab(r))$

$(\forall r \in \{<imp>\})\ ruletype(\_rlab(r)) = defeater$
$\rightarrow (\exists ar \in FA)\ N(ar) = ruleID(\_rlab(r))$

$RA = SA \cup DA \cup FA$

Rule circles are connected with the arrows representing rules, regardless their type:

$(\forall rc \in RC)\ (\forall ar \in RA)\ \ N(rc) = N(ar)$
$\rightarrow Out(rc) = ar \wedge In(ar) = rc$

The expressions $In(x)$ and $Out(x)$ denote pointers to the previous/next graph element, respectively.

As for the superiority relationship, it 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 (SUP attribute) with the inferior rule (INF attribute). The set of all superiority arrows is SRA.

$(\forall r \in \{<imp>\})\ (\forall sr \in superior(\_rlab(r))$
$(\exists ar \in SRA)\ SUP(ar) = r \wedge INF(ar) = sr$

Superiority arrows connect two rule circles:

$(\forall ar \in SRA)\ \ Out(SUP(ar)) = ar \wedge In(ar) = SUP(ar)$
$\wedge\ In(INF(ar)) = ar \wedge Out(ar) = INF(ar)$

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

$(\forall cp \in CP)\ (\exists ar \in CA)\ N(ar) = <cp, Body(cp)>$

where $<cp, Body(cp)>$ is a tuple that consists of the class pattern ID and the corresponding rule ID. Both are needed to uniquely identify such arrows, because the same class pattern can be re-used in the body of many rules.

Class patterns of the rule body are connected to rule circles as follows:

$(\forall\ ar \in CA)$

$(\forall\ rc \in RC)\ <cp,r>=N(ar) \wedge r=N(rc)$

$\rightarrow Out(cp) = ar \wedge In(ar) = cp \wedge In(rc) = ar \wedge Out(ar) = rc$

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:

$(\forall\ ar \in RA)\ (\forall\ cp \in CP)$

$Head(cp) = N(ar) \rightarrow In(cp) = ar \wedge Out(ar) = cp$

3.4. 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 *RS* that includes all the rules 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 (for example, see [12]). 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 and more comprehensible graph possible. The algorithm is displayed in Fig. 8.

```
str:=1
foreach cb∈CB_b do stratum(cb):=str
while |RS|≠0 do
   RuleTemp:=∅
   str:=str+1
   foreach R∈RS do
      if ((∀p∈premises(R) → stratum(class(p))<str) ∧
           (∃p'∈premises(R) ∧ stratum(class(p'))=str-1))
      then stratum(R):=str, RS:=RS-{R}, RuleTemp:=RuleTemp∪{R}
   foreach R∈RuleTemp do
      foreach p∈premises(R) do
            if stratum(class(p))=str-1
            then Type:=plain else Type:=expandable,
            in-arrow(R):=in-arrow(R)∪{<p,Type>},
            out-arrow(p):=out-arrow(p)∪{<R,Type>},
   str:=str+1
   CbTemp:=∅
   foreach R∈RuleTemp do
      if unknown(stratum(class(conclusion(R))))
      then stratum(class(conclusion(R))):=str,
           CbTemp:=CbTemp∪{class(conclusion(R))}
   foreach R∈RuleTemp do
      if type(R)=strictrule then Type:= strict
      else if type(R)=defeasible then Type:=defeasible
      else Type:=defeater,
      if class(conclusion(R))∈CbTemp
      then Orient:=plain else Orient:=dotted,
      out-arrow(R):=
            out-arrow(R)∪{<conclusion(R),Orient,Type>},
      in-arrow(class(conclusion(R))):=
            in-arrow(conclusion(R))∪{<R,Orient,Type>}
```

Fig. 8. The rule stratification algorithm

The algorithm aims at giving a left-to-right orientation to the flow of information in the graph; namely, the arcs in the digraph are directed from left to right, resulting in a less complex derived graph. The graph elements are "*stratified*", meaning that they are placed 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:

1. All the base class boxes are placed in stratum #1.
2. The algorithm enters a loop, consecutively assigning strata to rule circles and derived class boxes, incrementing each time the stratum counter by 1.
   a. In order for a rule circle to be assigned to a stratum, all its premises have to belong to previous strata, with at least one of them belonging to the immediately previous stratum.
   b. In order for a class box to be assigned to a stratum, it has to contain the conclusions of rules in the immediately previous stratum.

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 led to the desired rule conclusion in a previous stratum.

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.

3.5. Example of Using dl-RuleViz

This section outlines an example that could better illustrate the functionality of the algorithm described above. Suppose that we have the following rule base:

r_1: novel(X) → book(X)

r_2: book(X) ⇒ hardcover(X)

r_3: novel(X) ⇒ ¬hardcover(X)

r_4: novel(X), collectible(X,"yes") ⇒ rare(X,"yes")

r_5: novel(X),author(X,"Asimov"), price(Y), Y>18 ⇒ hardcover(X)

The first three rules were encountered in section 2.1, while rule r_4 reads as "*Collectible novels are considered rare books*" and rule r_5 reads as "*Novels by*

Asimov with a price greater than 18 are typically hard-covered". Also, three classes are needed in the example, as Table 1 indicates: one base class (`novel`) and two derived classes (`book` and `hardcover`).

Table 1. Classes included in the example

Base Class	`novel`
Derived Classes	`book, hardcover`

Table 2. Stratum assignments

stratum #1	`novel`
stratum #2	r_1, r_3, r_4, r_5
stratum #3	`book, hardcover`
stratum #4	r_2

After applying the algorithm, it comes up that four strata are needed to display all the graph elements. Table 2 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.

Fig. 9 displays the resulting graph, produced by dl-RuleViz, the rule base visualization module of DRREd. The implementation is compliant with the algorithm presented previously. 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, the user is navigated to the corresponding rule conclusion, applied in a previous stratum. In this case, the conclusion is the positive atomic formula highlighted in the figure.

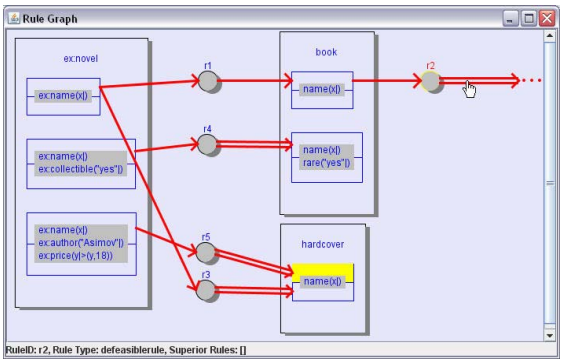


Fig. 9. Implementation of the visualization algorithm

4. Related Work

d-GRAPHER [10] 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 is the first system that offered visual development of d-graphs, adopting a representation that comprised the starting point for dl-RuleViz, 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.

On the other hand, there exists a variety of systems that implement rule representation and visualization, although, to the best of our knowledge, no modern system exists yet that can visually represent defeasible logic rules. Such an example is *VisiRule* [11], 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.

CPL (Conceptual Programming Language) [6] constitutes an effort to bridge the gap between Knowledge Representation and Programming Languages. CPL is a visual language for expressing procedural knowledge explicitly as programs. The basic notion are Conceptual Graphs, which are connected, multi-labeled, bipartite, oriented graphs and can express declarative and procedural knowledge, by defining object and action constructs. Particularly, the addition of visual language constructs (data-flow/flowchart) to Conceptual Programming allows the process of actions as data-flow diagrams that convey the procedural nature of knowledge within the representation. Both CPL and the dl-RuleViz underlying visual rule language are expressive; the latter, however, adds the flexibility and intuitiveness of defeasible reasoning to the graph.

5. Conclusions and Future Work

This paper argued that logic is currently the target of the upcoming efforts towards the realization of the Semantic Web vision and presented an algorithm for visualizing defeasible logic rule bases. dl-RuleViz, a tool that implements the proposed algorithm, was also briefly presented. For the representation of defeasible logic rules, the software utilizes directed graphs, which, however, find it difficult to associate data of a variety of types with the nodes and with the connections between the nodes in the graph. Trying to lever-

age this disadvantage, dl-RuleViz adopts a novel "enhanced" digraph approach that features distinct types of nodes for rules and literals and various distinct connection types for each defeasible logic rule type and the superiority relationship.

As for our future research goals, a variety of tasks still remain to be addressed. Potential improvements of dl-RuleViz and the visualization algorithm include enhancing the derived graph with negation-as-failure and variable unification, for simplifying the display of multiple unifiable class patterns. Expressive visualization of a defeasible logic rule base can then lead to proof explanations. By adding visual rule execution tracing, proof visualization and validation to the dl-RuleViz module, we can delve deeper into the Proof layer of the Semantic Web architecture, implementing facilities that would increase the trust of users towards the Semantic Web.

6. References

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

[2] Bassiliades N., Kontopoulos E., Antoniou G., "A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web", *Proc. RuleML-2005*, Galway, Ireland, Springer-Verlag, LNCS 3791, pp. 172-186, 2005.

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

[4] CLIPS Basic Programming Guide (v. 6.24), www.ghg.net/clips/CLIPS.html, last accessed: April 27, 2007.

[5] Diestel R., *Graph Theory* (Graduate Texts in Mathematics), 2nd ed. Springer, 2000.

[6] Hartley R., Pfeiffer H., "Visual Representation of Procedural Knowledge", *Proc. 2000 IEEE Int. Symp. on Visual Languages*, IEEE Computer Society, Washington DC, 2000.

[7] Kontopoulos E., Bassiliades N., Antoniou G., "Visualizing Defeasible Logic Rules for the Semantic Web", *Proc. 1st Asian Semantic Web Conf. (ASWC'06)*, Springer-Verlag, LNCS 4185, pp. 278-292, Beijing, China, 2006.

[8] Nute D., "Defeasible Reasoning". *Proc. 20th Int. Conf. on Systems Science*, pp. 470-477, IEEE Press, 1987.

[9] Nute D., Erk K., "Defeasible logic graphs: I. Theory", *Decis. Support Syst.*, 22(3), pp. 277-293, 1998.

[10] Nute D., Hunter Z., Henderson C., "Defeasible logic graphs: II. Implementation", *Decis. Support Syst.*, 22(3), pp. 295-306, 1998.

[11] Shalfield R., *VisiRule User Guide*, http://www.lpa.co.uk/ftp/4600/vsr_ref.pdf, 2005.

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

[13] W3C, Web Ontology Language (OWL) http://www.w3.org/2004/OWL/.