Deploying Defeasible Logic Rule Bases for the Semantic Web

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 is currently the target of the majority of the upcoming efforts towards the realization of the Semantic Web vision, namely making the content of the Web accessible not only to humans, as it is today, but to machines as well. Defeasible reasoning, a rule-based approach to reasoning with incomplete and conflicting information, is a powerful tool in many Semantic Web applications. Despite its strong mathematical background, logic, in general, and defeasible logic, in particular, may overload the user with tons of additional complex semantic relationships among data and metadata of the Semantic Web. To this end, a comprehensible, visual representation of these semantic relationships (rules) would help users understand them and make more use of them. This paper presents VDR-DEVICE, a defeasible reasoning system, designed specifically for the Semantic Web environment. VDR-DEVICE is an integrated development environment for deploying and visualizing defeasible logic rule bases on top of RDF Schema ontologies. The system consists of a number of subcomponents, which, though developed autonomously, are combined efficiently, forming a flexible framework. The system employs a defeasible reasoning system that supports direct importing and processing of RDF data and RDF Schema ontologies as well as a number of user-friendly rule base and ontology visualization modules.

Keywords: Semantic Web, Reasoning, Defeasible Logic, Visual Representation, Rule Editor, Rule Markup Languages

1. Introduction

The *Semantic Web* (SW) constitutes an effort to improve the current Web, by adding metadata to web pages and, thus, making the content of the Web accessible not only to humans, as it is today, but to machines as well [17]. This way, software agents for example, will be able to "understand" the meaning of the information available on the World Wide Web (WWW), resulting in better cooperation among agents as well as between agents and human users. The basic principle behind the SW initiative lies in organizing and inter-relating the available information, so that it can be utilized more effectively by a variety of distinct Web applications.

The SW has not yet acquired a concrete substance, but still remains a vision to a great extent, although various SW applications and software tools exist that can offer solutions to practical problems. Stephens in [58] provides a list with existing applications that perform integration of heterogeneous scientific data or optimisation of enterprise search and navigation, but also with potential applications, such as enhancing

the effectiveness of recruiting services or identifying patterns and insights in data. However, in order for SW technologies to be fully adopted, they must first cope with the problems of scalability, availability and reliability, all of which are fundamental issues in enterprises. After these issues have been tackled with, then end-users will begin to conceive the benefits that stem from the SW and realize the convenience and diversity of services it offers. Furthermore, the new type of data and metadata should not become an additional burden for the users who are already overwhelmed by information overload.

The mature steps towards accomplishing the Semantic Web vision have reached as far as the development of *ontologies* and *OWL*, the Web Ontology Language, which is currently the dominant standard in ontology encoding [14]. The upcoming efforts will be targeted at logic and proofs that are believed to posses a key role in assisting users towards eventually accepting the Semantic Web.

Defeasible reasoning [46], a member of the non-monotonic reasoning family, constitutes a useful tool in this context. It represents a simple, rule-based approach to reasoning not only with incomplete or changing but also with conflicting information. When compared to mainstream non-monotonic reasoning, the main advantages of defeasible reasoning are enhanced representational capabilities, coupled with low computational complexity. Defeasible reasoning can represent facts, rules and priorities and conflicts among rules. Such conflicts arise, among others, from rules with exceptions, which are a natural representation for policies and business rules (e.g. [54], [55]) and priority information is often available to resolve conflicts among rules.

Despite its strong mathematical background, logic, in general, and defeasible logic, in particular, may overload the user with tons of additional complex semantic relationships among data and metadata of the Semantic Web. To this end, a comprehensible, visual representation of these semantic relationships (rules) would help users understand them and make more use of them.

In this paper we present *VDR-DEVICE*, a system designed specifically for the SW environment. It is an integrated development environment for deploying and visualizing defeasible logic rule bases on top of RDF Schema ontologies. VDR-DEVICE consists of a number of subcomponents, which, though developed autonomously, are combined efficiently, forming a flexible framework. Although early versions of the various VDR-DEVICE visualization sub-components have already been presented in previous work of ours ([12], [37], [38]), the focus of this paper is to study both the integration into a unique system and the interaction between the various subcomponents.

At the core of the system lies a defeasible logic reasoner that processes rule bases, expressed in textual and XML form, more thoroughly presented in [11]. The reasoning core imports and processes RDF data and RDF Schema ontologies. Although the reasoning engine was successfully applied to a number of agent-based scenarios ([7], [57]), as the size and complexity of a rule base increase, it becomes more difficult for users to realize and comprehend dependencies among data and rules and the overall underlying structure of the rule base. This is why the system is also comprised of (a) a graphical defeasible logic rule base editor, an early version of which can be found in [12], (b) a visual RDF Schema ontology editor, a design and early implementation of which was presented in [38], and (c) a visual tool that produces graph-based representations of defeasible logic rule bases, whose representational schema (at least an early version of it) can be found in [37]. Thus, VDR-DEVICE is not a task-specific tool, namely, it does not aim at a specific layer/technology of the SW architecture. Instead, its functionality covers most of the SW layers, starting from content representation

(*XML*) and reaching logic (*inference*), while the next steps for the improvement of the system are planned to encompass *proofs* as well [10].

During the various development phases of the system, a number of requirements and specifications were being designated. The primary requirement, regarding the reasoning engine, was to support defeasible reasoning. As a consequence, a significant requirement for the rule editor and the rule base visualization module was to adopt a representation schema, which would prove intuitive and easy to apply. Moreover, a secondary requirement for the rule and RDF Schema ontology editors is to prevent users from syntactic and semantic errors during the development of rule bases and ontologies respectively.

The requirements mentioned above were taken into account during the design and implementation of VDR-DEVICE and the results are described in this paper, the rest of which is organized as follows: Section 2 describes the basic characteristics of defeasible logic and outlines some of the most usual cases for applying defeasible reasoning in the Semantic Web. Section 3 presents our approach for visualizing defeasible logic rules, based on digraphs, while the next section describes the VDR-DEVICE system in depth, covering in detail all its subcomponents. Section 5 presents a user evaluation, performed on two VDR-DEVICE components, while the next section discusses related work paradigms, followed by the conclusions and ideas for future work.

2. Defeasible Logics

This section briefly describes the basic characteristics of defeasible reasoning and the motivation for utilizing defeasible reasoning in the Semantic Web environment.

2.1. Basic Characteristics

The root of defeasible logics lies on research in knowledge representation and particularly on *inheritance networks*. Defeasible logics can be seen as inheritance networks, expressed in a logical rule language. In fact, they are the first non-monotonic reasoning approach, designed from its beginning to be realisable.

Being non-monotonic, defeasible logics deal with potential conflicts (inconsistencies) among knowledge items. Thus, contrary to usual logic programming systems, they contain classical negation. However, they can also deal with *negation-as-failure* (NAF), the other type of negation typical of non-monotonic logic programming systems; in fact, Wagner argues that the Semantic Web requires both types of negation [62]. In defeasible logics it is often assumed that NAF is not included in the object language (the language, in which rules are expressed); however, it can be easily simulated when necessary [6]. Thus, NAF can be used in the object language and the original knowledge can be transformed to logical rules without NAF, exhibiting the same behaviour.

Conflicts among rules are indicated by a conflict among their conclusions. These conflicts are of local nature, meaning that they only arise between rules with conflicting heads, contrary to other more mainstream non-monotonic approaches, such as Reiter's default logics [53], where a conflict may involve several defaults, a fact that imposes computational cost [26], [36]. The simplest case is that one conclusion is the negation of the other. The more complex case arises when the conclusions have been declared to be mutually exclusive, a very useful representation feature in practical applications. Furthermore, while default logics attempt to provide an overview of all

possible worlds, defeasible logics are sceptical in the sense that conflicting rules do not fire. Thus, consistency of drawn conclusions is preserved. Priorities on rules may be used to resolve some conflicts among rules. Priority information is often found in practice, and constitutes another representational feature of defeasible logics.

The logics take a pragmatic view and have low computational complexity [40]. This is not only achieved through the facts that they are sceptical (avoid the computation of alternative extensions) and that the conflicts are local, both of which were mentioned above, but also through:

- the absence of disjunction;
- the use of unidirectional rules;
- the local nature of priorities; only priorities between conflicting rules are used, as opposed to systems of formal argumentation, where often more complex kinds of priorities (e.g. comparing the strength of reasoning chains) are incorporated.

Relaxing any of these restrictions would jeopardise one of the key advantages of defeasible logic, namely its polynomial complexity. Clearly, these restrictions come at the price of reduced expressive power, compared to other non-monotonic reasoning approaches. For example, defeasible logics cannot be used for applications in which alternative models or credulous reasoning are required, nor for solving NP-hard problems (for the latter we would need a representation of exponential size). However, despite its conceptual simplicity, defeasible logic appears effective in the representation and execution of a variety of practical problems, including agent negotiation [57], semantic brokering [7], modelling contracts [27], [32], and modelling business rules [2]. Interestingly, these are some of the most promising application problems that are expected to benefit from the use of Semantic Web techniques. Thus, it is interesting to study the deployment of defeasible reasoning within the Semantic Web framework, the main theme of this paper.

2.2. Motivation for Applying Non-monotonic Reasoning in the Semantic Web

As mentioned before, defeasible reasoning is a non-monotonic method for reasoning with incomplete and conflicting information. In this section we outline reasons why this kind of approach is useful in the setting of the Semantic Web.

2.2.1. Reasoning with Incomplete Information

Business rules often have to deal with incomplete information [2]: in the absence of certain information some assumptions have to be made that lead to conclusions not supported by classical predicate logic. In many applications on the Web such assumptions must be made, because other players may not be able (e.g. due to communication problems) or willing (e.g. because of privacy or security concerns) to provide information. This is the typical case for applying non-monotonic knowledge representation and reasoning.

2.2.2. Rules with Exceptions

Rules with exceptions are a natural way of representation for policies and business rules. And priority information is often implicitly or explicitly available to resolve conflicts among rules. Potential applications include security policies [8], business rules [2], e-contracting [27], brokering [7] and agent negotiations [28].

2.2.3. Default Inheritance in Ontologies

Default inheritance is a well-known feature of certain knowledge representation formalisms. Thus, it may play a role in ontology languages, which currently do not support this feature. Grosof and Poon present some ideas for possible applications of default inheritance in ontologies [32]. A natural way of representing default inheritance is rules with exceptions plus priority information. Thus, non-monotonic rule systems can be utilized in ontology languages.

2.2.4. Ontology Merging

When ontologies from different authors and/or sources are merged, inconsistencies and contradictions arise naturally (for example see [35]). Moreover, in domains such as legal reasoning, ontologies may be defeasible, that is open to potential inconsistencies, by their very nature. Predicate logic based formalisms, including all current Semantic Web languages, cannot cope with inconsistencies. If rule-based ontology languages are used (e.g. DLP [31]) and if rules are interpreted as defeasible (i.e. they may be prevented from being applied even if they can fire), then we arrive at non-monotonic rule systems. More generally, when rules (e.g. policies or business rules) are merged conflicts may arise easily, and a mechanism for reasoning with such conflicts is valuable; conflicting rules arise naturally in areas such as personalization (selection of what to show next), security (weighting rules for and against providing access to certain information), negotiations etc. Assuming that an automated ontology merging and not display of several alternatives is desired, a skeptical approach, as adopted by defeasible reasoning, is sensible, because it does not allow for contradictory conclusions to be drawn. Moreover, priorities may be used to resolve conflicts among rules, based on knowledge about the reliability of sources or on user input). Thus, non-monotonic rule systems can support ontology integration.

2.3. Defeasible Logic - Syntax and Operational Semantics

A *defeasible theory* D (i.e. a knowledge base or a program in defeasible logic) consists of three basic ingredients: a set of facts (F), a set of rules (R) and a superiority relationship (>). Therefore, D can be represented by the triple (F, R, >). We assume a function-free first-order language.

In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters. *Strict rules* are denoted by $A \rightarrow p$, where A is a set of literals and p a (positive or negative) literal, 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, written formally, would become: r_1: novel(X) → book(X).

Contrary to strict rules, *defeasible rules* 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*").

Defeaters, denoted by $A \rightsquigarrow p$, are rules that do not actively support conclusions, but only defeat conflicting defeasible conclusions, by producing evidence to the contrary. A defeater example is: r_2': cheap(X) ⇝ ¬hardcover(X), which reads as: "*Cheap books might not be hard-covered*". This defeater can defeat, for example, rule r_2 mentioned above.

Finally, the *superiority relationship* among the rule set R is an acyclic relation > on R, that is, the transitive closure of > is irreflexive. Superiority relationships are used, in order to resolve conflicts among rules. For example, given the defeasible rules r_2 and r_3, no conclusive decision can be made about whether a novel is eventually hardcovered 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 the novel is not hardcover. In this case rule r_3 is called *superior* to r_2 and r_2 *inferior* to r_3.

As stated earlier, the notion of *negation-as-failure* (NAF) plays an important role in defeasible reasoning, since it raises the level of expressivity over logical negation, offering the ability to describe exceptions or to describe exhaustive searches over the current knowledge base. If, for example, we wish to find the cheapest book in our library, then we should formulate rule

```
r': book(X), price(X,Y), NOT(book(Z), Z≠X, price(Z,W), W<Y)
      ⇒ cheapest(X).
```

Rule r′ states that if a book with a specific price exists and the existence of a book with a lower price *cannot be confirmed*, then the first book is considered the cheapest. According to NAF, this "lack of knowledge", regarding a cheaper book in the specific example, is considered a failure.

Finally, another important element of defeasible reasoning is the notion of *conflicting literals*. In applications literals are often considered to be conflicting and at most one of a certain set should be derived. An example of such an application is price negotiation, where an offer should be made by the potential buyer. Possible offers can be determined by several rules, whose conditions may or may not be mutually exclusive. All rules have a positive `offer` literal in their head, since all rules try to make an offer. However, *only one offer* literal should be concluded; thus, only one of the rules should prevail, based on superiority relations among them. In this case, the conflict set is:

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

The conflict set for the literal `offer(x,y)` contains the negation of the literal along with all the other offers that are different from `offer(x,y)`. An example in this setting is a book auction, where the following two rules make an offer for a novel, based on the potential buyer's requirements. However, the second one is more specific and its conclusion overrides the conclusion of the first one.

```
p: novel(X),price(X,Y),Y≤15 ⇒ offer(X,15)
q: novel(X),price(X,Y),Y≤15,author(X,"Asimov") ⇒ offer(X,20)
q > p
```

According to defeasible logic proof theory [4], in order to show that a conclusion *C* is provable defeasibly there are two choices: (1) to show that *C* is already definitely provable, using a strict rule; or (2) to show that there is a strict or defeasible rule with head *C* whose body literals have been defeasibly proven and there are no possible "attacks", that is, reasoning chains in support of ¬*C*. Formally, we must show that ¬*C* is not definitely provable. Also we must consider the set of all rules which are not known to be inapplicable and which have head ¬*C* (here we consider defeaters, too, whereas they could not be used to support the conclusion *C*). Essentially each such rule attacks the conclusion *C*. For *C* to be provable, each attacker must be counterattacked by another rule with head *C* 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 *C* must be counterattacked by a stronger rule.

Conclusively, defeasible logic has been studied extensively from the theoretical perspective. In this paper we report on a development environment for defeasible logic, embedded in semantic web technologies, so the focus is different. However, we summarize here the main results from previous works:

- A full proof theory was presented in [4].
- The formal properties of the proof theory, including consistency and coherence results, were presented in [4].
- Relationships to other logic programming systems were presented in [6] and [5].
- An analysis of computational complexity is found in [40].
- Model-theoretic and argumentation semantics, with associated soundness and completeness results, are found in [39] and [29], respectively.
- Variations of defeasible logics were discussed in [3].

3. Visualizing Defeasible Logic Rules

Although defeasible logic is expressive, when the rule base increases in size and complexity, it is useful to have the ability to visualize and comprehend dependencies and competitions among data and rules, as well as visually identifying and differentiating among the various rule types. So, end users might often consider a graphical trace and an explanation mechanism very beneficial. Overall, in the case of complex rule bases, visualization mechanisms are particularly useful, since they can better assist in clarifying the underlying structure of the rule base.

In this work we propose the use of *directed graphs* [34] (or *digraphs*, as they are usually referred to) in the graphical representation of defeasible logic rules. To an extent, our approach is based on the methodology presented by Nute in [47], who applies *d-graphs* (defeasible logic graphs), for visualizing a set of defeasible logic rules. However, the methodology we adopt adds a variety of extra features that offer expressive power to the graph.

Therefore, in an attempt to leverage the graphs' inability to associate data of a variety of types with the vertices and edges, we propose distinct vertex and edge types. Thus, the digraphs in our approach contain two kinds of vertices ($V=L\cup R$):

- Literals (L), represented by rectangles, called "*atomic formula boxes*";
- Rules (R), represented by circles.

Edges ($E=RE\cup CE$) are either rule edges (RE) or condition edges (CE). There are three *rule* edge types ($RE=S\cup D\cup F$), one for each of the three rule types of defeasible reasoning (strict rules S, defeasible rules D, defeaters F), similarly to [47]. This results in rules of different types being represented more distinctively.

Rule edges connect a rule vertex with a rule conclusion vertex, namely a derived literal:

$$RE = \{(x, y) \mid x \in R \wedge y \in L\}$$

Each rule vertex must have one and only one rule edge:

$$(\forall x), x \in R \rightarrow (\exists y), y \in L \wedge (x, y) \in RE$$

$$(\forall x)(\forall y), (x, y) \in RE \rightarrow (\neg \exists y'), y' \in L \wedge y' \neq y \wedge (x, y') \in RE$$

There is only one *condition* edge type CE that connects a literal vertex with a rule vertex:

$$CE = \{(x, y) \mid x \in L \wedge y \in R\}$$

The same literal vertex can participate in multiple condition edges, meaning that the same atomic formula can appear in the condition of multiple rules. Furthermore, the same rule vertex can participate in multiple condition edges, meaning that a rule can have multiple atomic formulas in its condition. When this happens the condition of the rule is satisfied when ALL atomic formulas are simultaneously satisfied (logical conjunction).

Notice that the defeasible rule graph is bipartite, since each edge, regardless of its type, connects either a rule vertex to a literal vertex or vice versa.

The visual depiction of the rules, presented previously in section 2.3, is illustrated below. More specifically, Fig. 1 displays rule r_1 (strict rule), Fig. 2 displays rules r_2 and r_3 (defeasible rules) and Fig. 3 shows rule r_2' (defeater).

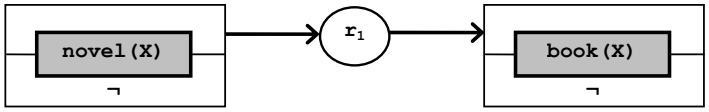


Fig. 1. Visual representation of strict rule r_1

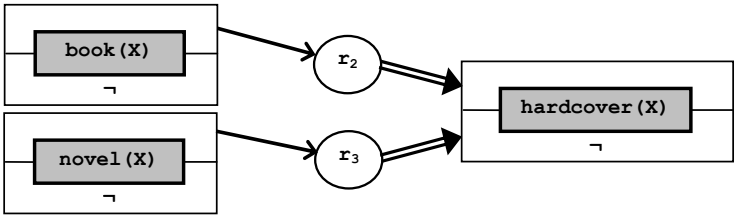


Fig. 2. Visual representation of defeasible rules r_2 and r_3

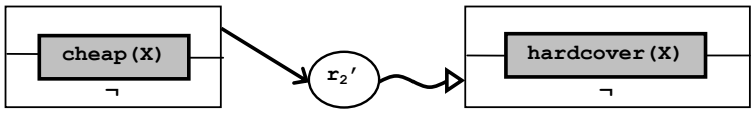


Fig. 3. Visual representation of defeater r_2'

As a notational convention, each atomic formula box (i.e. each literal vertex) is enclosed in a *literal box*, which consists of two adjacent atomic formula boxes, where the upper one represents a positive and the lower one represents the corresponding negative atomic formula. This way the two contradicting atomic formulas are depicted together but clearly separated. The set of all (normal) literal boxes *LB* is formally defined as a set of pairs of contradicting atomic formula boxes (or literal vertices):

$$LB = \{(x, \neg x) \mid x \in L^+\}$$

The set L^+ is the set of all positive (or unsigned) atomic formulas. The set L^- is the set of all negative (signed) atomic formulas. These two sets are disjoint.

$$L^- = \{\neg x \mid x \in L^+\},\ L^+ \cap L^- \equiv \varnothing$$

For example, in Fig. 2, the head of rule r_2 is positive, while that of rule r_3 is negated; this is determined by the atomic formula box (upper/lower), on which the arrow commencing from the rule circle "lands".

Arguments (variables or constants) can be incorporated inside the literal box, just after the predicate name of each literal box. We call the set of all arguments for each literal box, an *argument pattern*. Notice that a rule graph stands for one rule. However, since rules usually have variables, it also stands for the set of all possible instantiations of the variables. Therefore, the same rule graph can be interpreted both as a

schema and as a set of rule instantiations. This is also true for all the extensions and variations of the atomic formula boxes found later in the paper.

Finally, there is one more edge type *SE* for the superiority relationship. Fig. 4 shows the superiority relationship `r3 > r2` presented in section 2. Notice that this edge type connects only rule vertices and it is orthogonal to the bipartite defeasible rule graph.

$$SE = \{(x, y) \mid x \in R \wedge y \in R\}$$

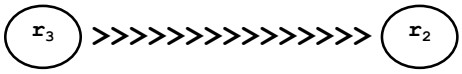


Fig. 4. Visual representation of the superiority relationship $r_3 > r_2$

3.1. Representing Conditions

So far we have demonstrated how rules are represented by interconnecting literal boxes with rule vertices and, also, how arguments of literals are presented, either being variables or constants. However, variables are usually associated with simple conditions (e.g. X > 4), which theoretically could be represented as predicates, but practically it is more convenient to consider them more closely related to the literal where the corresponding variable appears as an argument for the first time.

Simple conditions associated with any of the variables of a literal also appear inside the literal box. However, since there can be many conditions, each one of them appears on a separate line (called *condition pattern*) below the literal. For example, if the fragment `price(X,Y),Y>15` appears in a rule condition, it can be represented as in Fig. 5.

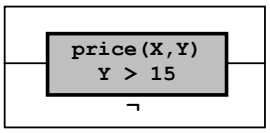


Fig. 5. Representing simple conditions on variables

3.2. Visualizing NAF

Negation-as-failure or default negation (NAF) is represented with the help of *backward hyperarcs*, or simply *B-arcs* [25]. A B-arc is a hyperarc *he*=(T(*e*),H(*e*)) (*he*∈*HE*) with |H(*e*)|=1; in other words, it's a directed hyperedge, with a tail *T* consisting of one or more vertices (|T(*e*)|>=1) and a head *H* consisting of a single vertex. The NAF hyperedge connects multiple literal vertices with one rule vertex and is actually considered a condition edge of a special type.

$$HE = \{(X, \{y\}) \mid X \subset L \wedge y \in R\}$$

Of course, the definition of condition edges, given in section 3 above, must be extended to handle hyperdges as well.

$$CE = \{(x, y) \mid x \in L \wedge y \in R\} \cup HE$$

The above definition suggests that the condition edges are either normal ones or hyperdges, the latter denoting NAF in the rule condition. Furthermore, notice that a hyperedge can have a tail with a single vertex; however, this is not equivalent to a normal edge. To exemplify this, the difference between the two rules below is that for rule `r1` a normal condition edge (`b`,`r1`) is drawn between literal vertex `b` and rule ver-

tex r_1, whereas for rule r_2 a hyperedge ({d},{r_1}) is drawn between literal vertex d and rule vertex r_2.

r_1: a, b → c

r_2: a, not(d) → c

The meaning of the NAF hyperedge is that when ALL atomic formulas of the literal vertices that participate in the tail of the hyperedge are satisfied, then the condition of the rule vertex that participates in the head of the hyperedge is NOT satisfied. When at least one of the atomic formulas of the tail is not satisfied, then the satisfaction of the condition of the rule depends on the rest of the condition edges that it participates.

Fig. 6 displays rule r′ from section 2.3, represented as a digraph, featuring a hyperedge. As can be observed, the NAF arc in the specific example has a tail that consists of two vertices/literal boxes (|T(*E*)|=2) and a head that consists of just one vertex/rule circle (|H(*E*)|=1).

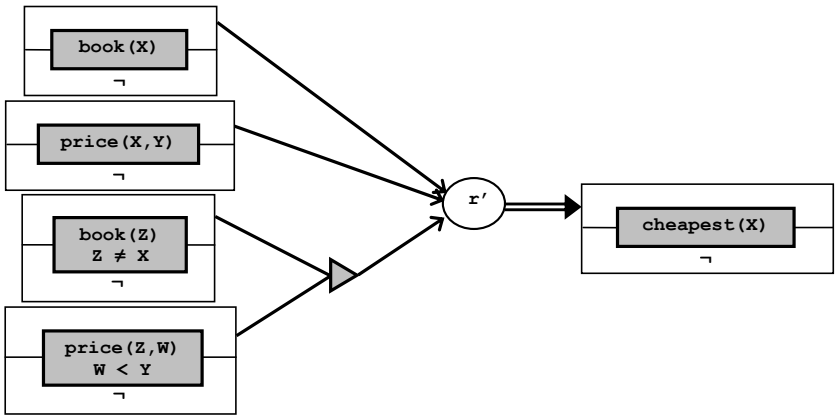


Fig. 6. Representation of negation-as-failure

3.3. Representing Conflicting Literals

For the representation of conflicting literals, consider rules p and q, presented earlier in section 2.3, which both produce the same literal type as a conclusion:

p: novel(X),price(X,Y),Y≤15 ⇒ offer(X,15)

q: novel(X),price(X,Y),Y≤15,author(X,"Asimov") ⇒ offer(X,20)

The graph drawn by the two rules is depicted in Fig. 7 and, as can be observed, both rules produce the same result type, which is included in a *multi-literal truth box*. The multi-literal truth box consists of multiple adjacent (positive) atomic formula boxes, all of which include the same literal type, although the argument patterns can potentially be dissimilar. At most one of the literals must be true and superiority relationships (in the specific example q > p) can determine the priorities among the rules.

Formally, multi-literal truth boxes are sets of conflicting literals of the same "sign" (either all positive or all negative):

$$MLB = \{mlb \mid mlb \in \wp(L^+) \vee mlb \in \wp(L^-)\}$$

Notice that the definition of literal boxes, given above in section 3, must be extended to cover multi-literal truth boxes, as well. Therefore, the set of literal boxes *LB* consists of *normal literal boxes* (*NLB*), the ones that couple together positive and

negative atomic formulae and have been presented above, and multi-literal truth boxes (*MLB*), the ones presented in this subsection:

$LB = NLB \cup MLB$, where $NLB = \{\{x, \neg x\} \mid x \in L^+\}$

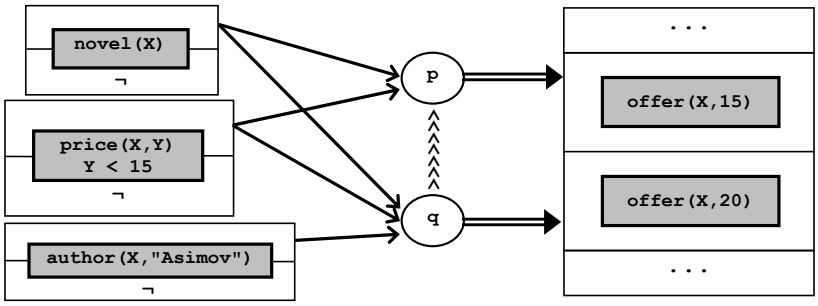


Fig. 7. Representation of conflicting literals as a digraph

Notice that the definition of normal literal boxes has been slightly altered from pairs to sets of two members, because elements of set LB should be of the same type.

3.4. Grouping Literal Boxes Together

A certain predicate, say `price`, can appear many times in a rule base, in many rule conditions or even rule conclusions (if it is not a base predicate, i.e. a fact). All literal boxes of the same predicate can be grouped together (as a notational convention) so that the user can visually comprehend that all such boxes refer to the same set of literals. In order to achieve this, we introduce the notion of a *predicate box*, which is simply a container for all the literal boxes that refer to the same predicate. Predicate boxes are labelled with the name of the predicate. Furthermore, the literal boxes contained inside the predicate box "lose" the predicate name, since the latter is located at the top of the predicate box. Such a literal box, which appears inside a predicate box and expresses conditions on instances of the specific predicate extension, is called *predicate pattern*.

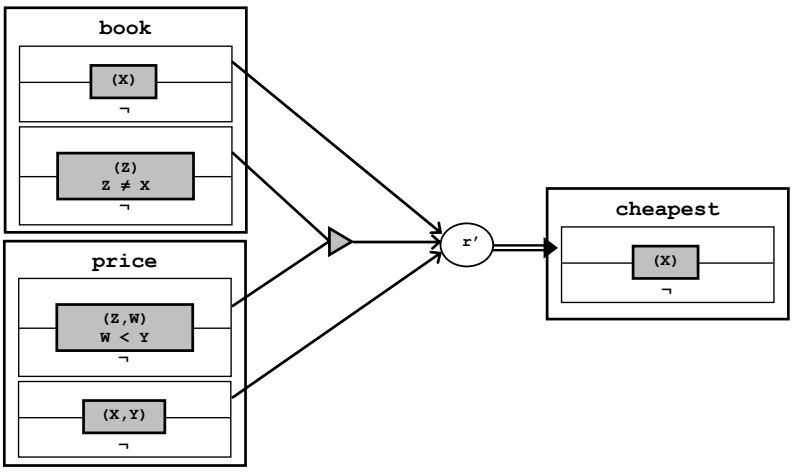


Fig. 8. Predicate box and predicate patterns

For example, the literal boxes in Fig. 6 can be grouped inside predicate boxes, according to the predicate name. Fig. 8 displays the resulting representation. Notice that, in general, each predicate pattern must contain exactly one argument pattern and zero, one or more condition patterns.

Notice that since the same predicate pattern can occur in multiple rule conditions, there might be multiple similar predicate pattern boxes. Since the intention of our visual representation scheme is compactness, only a single predicate pattern box appears and it participates in multiple condition edges. Furthermore, there might be predicate patterns that are syntactically different, but they can be made similar through a unification process. The theoretical part of our representation scheme dictates that these predicate pattern boxes should be drawn separately because they involve different variable names in different rules; therefore, if we omit them the user could not understand how certain variables are used in a rule condition without being shared with other condition patterns. However, in the practical part of our work we keep just one of these unifiable patterns and when the user focuses on a certain rule, the variables of the actual predicate pattern are visualized on-the-fly. Details on the unification algorithm, as well as the literal box grouping, can be found in section 4.3.3.3.

4. The VDR-DEVICE System

VDR-DEVICE [12] is a flexible integrated development environment for deploying defeasible logic rule bases on top of RDF Schema ontologies and consists of two primary subsystems:

a. *the reasoning system* that performs inference on RDF metadata, using defeasible logic rules and produces the results, and
b. *the graphical front-end* that acts as the shell of the core reasoning system and embodies a variety of development and representation tools.

The following section illustrates the overall system functionality, while subsequent sections describe comprehensively the two subsystems as well as the various modules they encompass.

4.1. VDR-DEVICE Functionality

Fig. 9 displays a sequence diagram, describing the functionality of VDR-DEVICE. The system accepts as input the address of a defeasibe logic rule base (program, written in the RuleML-like syntax of VDR-DEVICE – see section 4.2.2), which is created or loaded with the help of the DRREd rule editor, described in the following sections.

The rule base contains only rules; the facts for the rule program are (input) RDF documents, whose addresses are declared in the rule base header. The RDFSbuilder module, also part of the VDR-DEVICE graphical front-end, can be applied in creating the RDF Schema for the facts loaded. Nevertheless, users are not obliged to utilize RDFSbuilder, since they can exploit whichever RDF Schema editor they feel comfortable with.

The rule base is submitted and the designated facts are downloaded by DR-DEVICE [10], the core reasoning module of the system, and the inference process commences. The rule conclusions are materialized inside DR-DEVICE as objects and the instances of designated derived classes are exported as an RDF document, which includes the RDF Schema definitions for the exported derived classes and those instances of the exported derived classes, which have been proved, either positively or negatively, either defeasibly or definitely.

The user can access the results through a web browser or through specialized software that can customize the visualization. Notice that DR-DEVICE can also provide

explanations about non-proved objects, i.e. objects that are not proved neither definitely nor defeasibly.

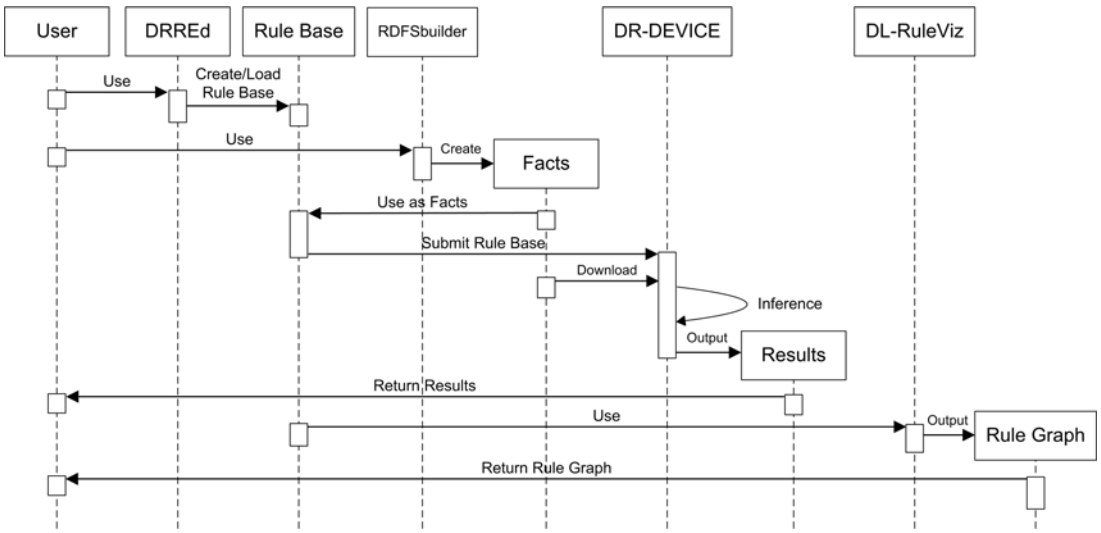


Fig. 9. VDR-DEVICE functionality

Finally, the system also offers the capability of visualizing the rule base, by automatically producing a stratified directed rule graph with the help of the DL-RuleViz tool, also described in a subsequent subsection.

4.2. DR-DEVICE Architecture

As mentioned earlier, DR-DEVICE [11] comprises the core reasoning system of VDR-DEVICE and consists of two primary components (Fig. 10): The *RDF loader/translator* and the *rule loader/translator*. The rule base is initially submitted to the *rule loader*, which transforms it into the native CLIPS-like syntax through an XSLT stylesheet and the resulting program is then forwarded to the *rule translator*, where the defeasible logic rules are compiled into a set of CLIPS production rules [19]. This is a two-step process: First, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic deductive rule language of R-DEVICE [13], using the translation scheme described in [10]. Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in [13]. All compiled rule formats are also kept in local files (structured in project workspaces), so that the next time they are needed they can be directly loaded, improving speed considerably (running a compiled project is up to 10 times faster).

Meanwhile, the *RDF loader* downloads the input RDF documents (facts), including their schemas, parses them, using the ARP parser, contained in the *Jena Semantic Web Framework* [44] and translates RDF descriptions into CLIPS objects, according to the RDF-to-object translation scheme in [13]: the inference engine of CLIPS performs the reasoning by running the production rules and generates the objects that constitute the result of the initial rule program. The compilation phase, based on the meta-program of [42], guarantees correctness and completeness of the reasoning process according to the operational semantics of defeasible logic.

Finally, the result-objects are exported to the user as an RDF/XML document through the RDF extractor [13]. The RDF document includes the instances of the ex-

ported derived classes, which have been proved, either positively or negatively, either defeasibly or definitely.

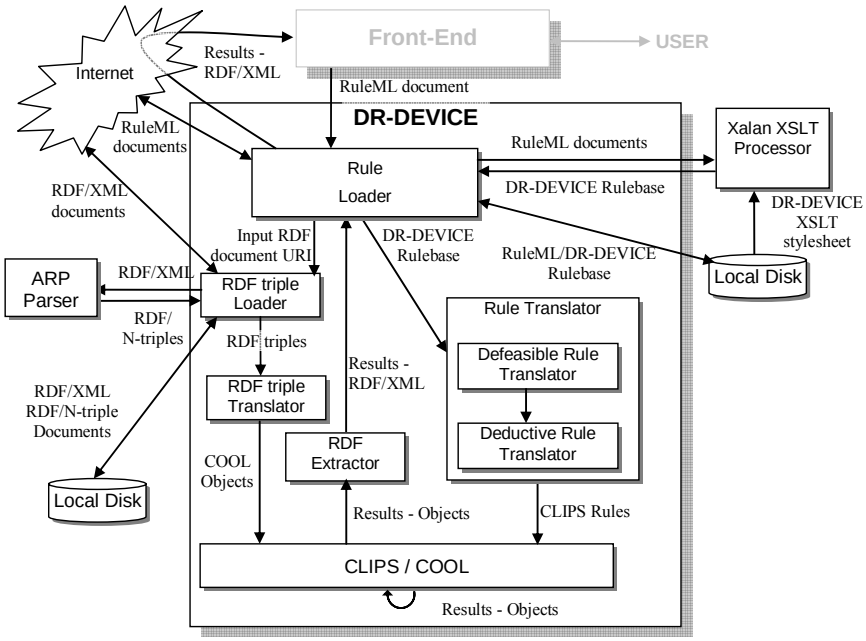


Fig. 10. The architecture of the DR-DEVICE defeasible reasoning system

4.2.1. The Object-Oriented RDF Data Model

The DR-DEVICE system employs an object-oriented view of the RDF data model, where properties are treated as normal object attributes, encapsulated in resource objects. This way, properties of resources are not scattered across several triples, resulting in increased query performance due to less joins [13].

```
<rdfs:Class rdf:about="&books;novel" rdfs:label="novel">
  <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="&books;name" rdfs:label="name">
  <rdfs:domain rdf:resource="&books;novel"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:about="&books;collectible"
      rdfs:label="collectible">
  <rdfs:domain rdf:resource="&books;novel"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:about="&books;author"
     rdfs:label="author">
  <rdfs:domain rdf:resource="&books;novel"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:about="&books;price" rdfs:label="price">
  <rdfs:domain rdf:resource="&books;novel"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>

<books:novel rdf:about="&books_ex;novel_1">
  <books:name>I,Robot</books:name>
  <books:author>Asimov</books:author>
  <books:price rdf:datatype="&xsd;integer">25</books:price>
  <books:collectible>yes</books:collectible>
</books:novel>
```

(a)

```
(defclass books:novel
  (is-a rdfs:Resource)
  (multislot books:name (type LEXEME))
  (multislot books:collectible (type LEXEME))
  (multislot books:author (type LEXEME))
  (multislot books:price (type INTEGER))
)
```

(b)

```
[books:price] of rdf:Property
(uri books:price)
(source rdf)
(rdfs:isDefinedBy)
(rdf:type [rdf:Property])
(rdf:value)
(rdfs:comment)
(rdfs:label "price")
(rdfs:seeAlso)
(rdfs:domain [books:novel])
(rdfs:range [xsd:integer])
(rdfs:subPropertyOf)
```

(c)

```
[books_ex:novel_1] of books:novel
(books:name "I,Robot")
(books:author "Asimov")
(books:price 25)
(books:collectible "yes")
```

(d)

Fig. 11. (a) RDF document excerpt, describing the "`novel`" class and its properties, as well as an instance of the class ("`novel_1`"), (b) definition of class "`novel`" in COOL, (c) transformation of an RDFS property into a COOL object, (d) COOL object, representing the novel instance in (a)

For example, Fig. 11(a) describes an RDF Schema class ("novel") with four properties ("name", "collectible", "author", "price") and a novel instance ("novel_1"). The definition of class "novel" in COOL [19] is shown in Fig. 11(b). Properties that have this class as their domain have been made slots for that class. Furthermore, each property has been made an instance of the rdf:Property class (e.g., property "price" in Fig. 11(c)), while Fig. 11(d) shows the COOL object that corresponds to instance "novel_1" in Fig. 11(a).

4.2.2. The Defeasible Logic Language

DR-DEVICE supports two types of syntax for defeasible logic rules: a native CLIPS-like syntax and a RuleML-compatible one. Here we focus solely on the latter, since the rule editor of the system allows the expression of rules only in this syntax. While the RuleML-like syntax utilizes as many features of the official RuleML as possible, several of the features of the rule language cannot be expressed by the existing RuleML DTDs and/or XML Schema documents. For this reason a new DTD (v. 0.85 compatible) and new XML Schema documents (up to v. 0.91 compatible) were developed, using the modularization scheme of RuleML, extending the nafneg DTD of RuleML, namely OO-Datalog with strong negation and negation-as-failure. A reference and DTD is included in [10]. However, the original DTD and XML Schema documents can also be found at http://lpis.csd.auth.gr/systems/dr-device.html, along with the system itself. Notice, that the system currently uses the v. 0.85 compatible DTD.

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

For example, Fig. 12 illustrates how defeasible rule r_3, presented earlier (in section 2.3), can be expressed in the RuleML-compatible language of the system.

```
<imp>
     <_rlab ruleID="r3" ruletype="defeasiblerule" superior="r2">
          <ind>r3</ind>
     </_rlab>
     <_head>
          <neg>
               <atom>
                    <_opr><rel>hardcover</rel></_opr>
                    <_slot name="name"><var>x</var></_slot>
               </atom>
          </neg>
     </_head>
     <_body>
          <atom>
               <_opr><rel href="books:novel"/></_opr>
               <_slot name="books:name"><var>x</var></_slot>
          </atom>
     </_body>
</imp>
```

Fig. 12. Representing rule r_3 (see section 2.3) in the DR-DEVICE RuleML-compatible syntax

The rules consist of a head and a body. The body is the condition of the rule, which, if satisfied, then the conclusion in the head of the rule is asserted. Actually, the rule condition corresponds to a complex query over the current set of objects in the knowledge base; if the result to the query is not the empty set, then the rule fires and a set of derived objects is asserted according to the template defined in the rule head. Each

atom in the rule body corresponds to a simple query over one predicate extension (in the case of predicate calculus) or one class, in the case of DR-DEVICE, where there are classes and objects, instead of predicates and facts. Notice that currently only logical conjunction in the rule body is supported, along with negation-as-failure.

The `rel` elements of the operator (`_opr`) elements of atoms correspond to class names, since atoms actually represent queries over CLIPS objects. The names of RDF classes are referred to through the `href` attribute of the `rel` element, in the atoms in the rule body, because they refer to classes defined elsewhere, while the names of the derived classes in the rule head appear as the content of the `rel` element. Atoms have named arguments, called *slots*, which correspond to object properties. Since RDF resources are represented as CLIPS objects, atoms correspond to queries over RDF resources of a certain class with certain property values. Superiority relations are represented as attributes of the superior rule (e.g. in Fig. 12 rule r_3 is superior to r_2). Negation is represented via a `neg` element that encloses an `atom` element.

There also exist `comp_rules` elements that declare groups of competing rules, which derive competing positive conclusions (*conflicting literals*). For example, rules `p` and `q` in Fig. 13 are competing over the conclusion `offer(X,Y)`, since at most one offer can be made in an offer case (see section 2.3). The constant `cr1` in the figure is just a label for this group of competing rules.

```
<comp_rules c_rules="p q">
  <_crlab> <ind>cr1</ind> </_crlab>
</comp_rules>
```

Fig. 13. Declaring groups of competing rules

Further extensions to the RuleML syntax, include function calls that are used either as constraints in the rule body or as new value calculators in the rule head. Additionally, multiple constraints in the rule body can be expressed through the logical operators: `_not`, `_and`, `_or`, whose semantics are similar to the CLIPS *connective constraints* [19]. Finally, the header of the rule base (`rulebase` element), includes a number of important parameters, implemented as attributes: `rdf_import` declares the input RDF file(s), `rdf_export` represents the RDF file that contains the exported results and `rdf_export_classes` represents the derived classes, whose instances will be exported in RDF/XML format. An example is shown in Fig. 14.

Further details on the RuleML defeasible logic rule language of DR-DEVICE, as well as its translation on a CLIPS-like notation and its execution as a set of deductive rules has been presented elsewhere [10] and does not need to be repeated here, since the main point of this work is to present the various visual interfaces to DR-DEVICE.

```
<rulebase rdf_import="http://lpis.csd.auth.gr/.../books.rdf#"
          rdf_export="http://lpis.csd.auth.gr/.../export-books.rdf"
          rdf_export_classes="hardcover book">
```

Fig. 14. The `rulebase` element and its attributes

4.3. The Graphical Front-end

The front-end of the system allows direct interaction with the software in a user-friendly manner. It is equipped with a number of tools that assist in developing defeasible logic rule bases, offering various representational aspects of the rule bases developed and modeling and creating RDF Schema ontologies. More specifically, these

tools are: (i) *DRREd*, a graphical defeasible logic rule base editor, (ii) *RDFSbuilder*, a visual RDF Schema ontology editor, and (iii) *DL-RuleViz*, a visual tool that produces graph-based representations of defeasible logic rule bases.

4.3.1. DRREd - The Graphical Rule Editor

Writing rules in RuleML can often be a highly cumbersome task; thus, the need for authoring tools that assist end-users in writing and expressing rules is imperative. VDR-DEVICE is equipped with *DRREd* (Defeasible Reasoning Rule Editor), a graphical rule editor that comprises the primary subcomponent of the system shell [12] and aims at enhancing user-friendliness and efficiency during the development of VDR-DEVICE RuleML documents.

The main window is composed of two major parts (Fig. 15): The left-hand-side panel displays the rule base in XML tree-like format. Users can navigate through the tree and can add elements to or remove elements from the tree, obeying to the DTD/XML Schema constraints. Furthermore, the operations allowed on each element depend on the element's meaning within the rule tree. However, no textual editing of the rule base is allowed, in order for the well-formedness to be preserved.

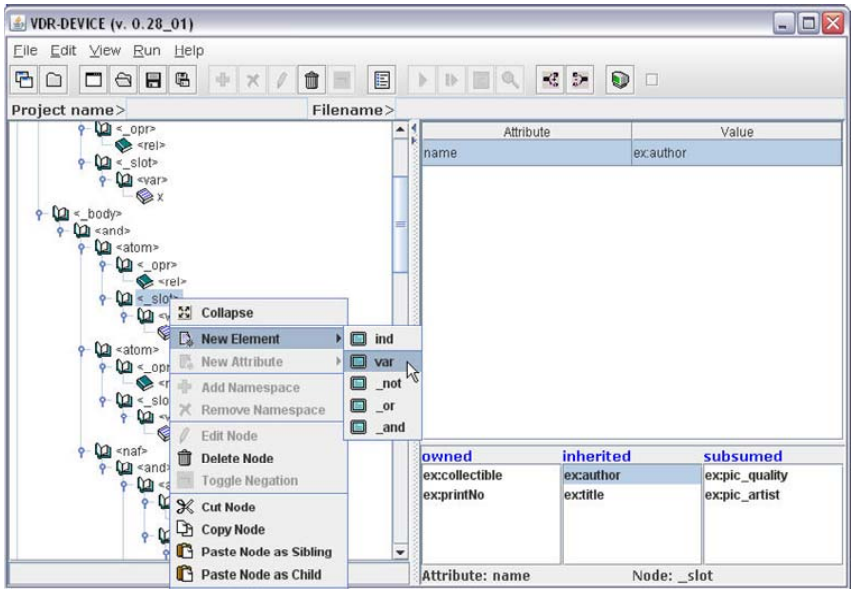


Fig. 15. The main window of the graphical rule editor

The right-hand-side panel shows a table, which contains the attributes that correspond to the selected tree node in the left panel (see section 4.2.2 for a reference to the available attributes). The user can perform editing functions on the attributes, by altering the value for each attribute in the panel that appears at the bottom-right of the main window, but the values that can be inserted depend on the chosen attribute.

The rationale behind the XML-tree representation of a rule base is based on the fact that RuleML is an XML application and, therefore, a tree representation seems intuitive. Furthermore, a tree can have its branches expanded and collapsed, better allowing users to focus on small or bigger parts of it.

The development of a rule base with DRREd is a process that depends heavily on the context, i.e. the node being edited. Thus:

- When a new element is added to the tree, all its mandatory sub-elements are also added. In case there are multiple alternative sub-elements, none is added; the user is

responsible to manually add one of them, by right-clicking on the parent element and choosing a sub-element from the pop-up menu that appears (Fig. 15).

- The `atom` element can be either negated or not. The wrapping/unwrapping of an `atom` element within a `neg` element is performed via a toggle button on the overhead toolbar.
- Since the core reasoning module (DR-DEVICE) features negation-as-failure (NAF), which is typical of non-monotonic logic programming systems (see section 2.1), DRREd also offers the capability of applying NAF on atoms. By right-clicking on an `and` element, users can add a `naf` child element that can encapsulate the atom(s) to be negated with NAF, as imposed by RuleML.
- The function names in a `fun_call` element are partially constrained by the list of CLIPS built-in functions. However, a custom user-defined function, which is unconstrained, can still be applied.
- Rule IDs uniquely represent a rule within the rule base; therefore, they are collected in a set and they are used to prohibit the user from entering duplicate rule IDs and to constrain the values of IDREF attributes (i.e. attributes that reference an existing ID, like the `superior` attribute that defines superiority relationships).

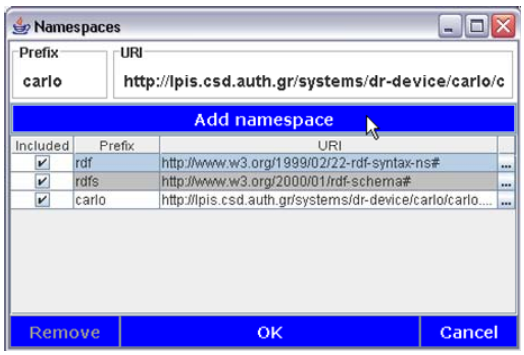


Fig. 16. The namespace dialog window

An important component of the editor is the *namespace dialog window* (NDW) (Fig. 16), which allows the user to determine which RDF/XML namespaces will be used by the rule base. Namespaces are treated as addresses of input RDF Schema ontologies that contain the vocabulary for the input RDF documents, over which the rules will be run. The namespaces that have been manually selected by the user to be included by the system are analyzed, in order for all the allowed class and property names for the rule base being developed to be extracted (see section 4.3.1.1). These names are then used throughout the authoring phase of the RuleML rule base, constraining the corresponding allowed names that can be applied and narrowing the possibility for errors on behalf of the user.

Namespaces can be manually entered by the user, through the NDW. Firstly, the system shows up in the NDW the namespaces contained in the input RDF documents (indicated by the `rdf_import` attribute of the `rulebase` root element). Notice that it is up to the user to include them or not as ontologies into the system. Furthermore, the system shows up only namespaces that actually correspond to RDF documents, i.e. it downloads them and finds out if they parse to triples. Next, they are checked for syntactic consistency and are confirmed to contain RDF Schema declarations. The user can also manually "discover" more namespaces, by pressing the "**…**" button next to each namespace entry. The system then downloads the namespace documents contained within this document and repeats the above namespace discovery procedure.

When it discovers a new namespace, not already contained in the NDW, it shows it up (unchecked).

Finally, users can examine all the exported results via a browser window, launched by the system. The user can also examine the execution trace of compilation and running, both at run-time and also after the whole process has terminated (Fig. 17).

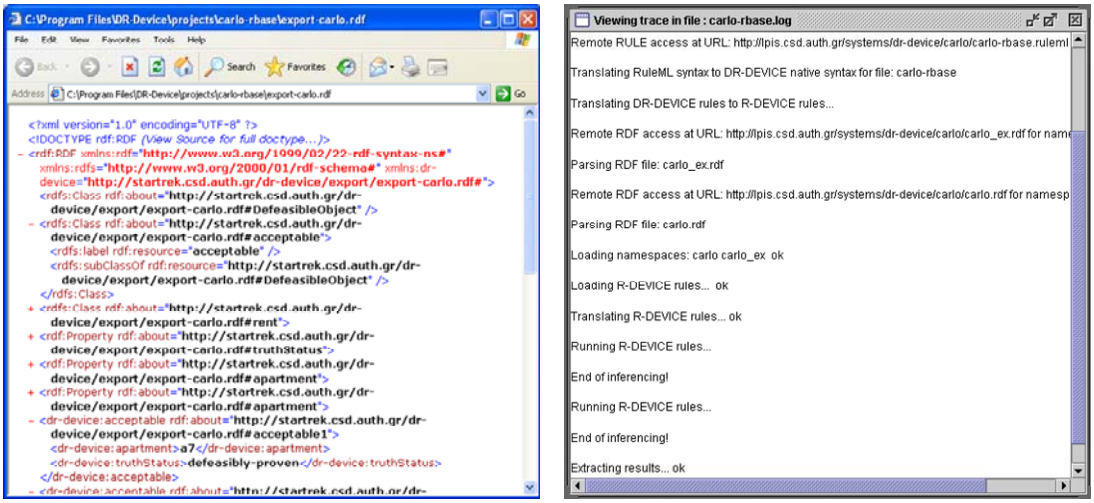


Fig. 17. VDR-DEVICE results and trace windows

In the following subsections, we present, in mathematical notation, how the information needed to support the functionality of DRREd is extracted from the RuleML document and the RDF Schema documents, discovered by DRREd and selected by the user in the NDW.

4.3.1.1. Parsing RDF Schema Ontologies for Classes and Properties

The RDF Schema documents contained in the NDW are parsed in order to collect all the definitions of base RDF classes and their properties, i.e. the vocabulary contained in the imported ontologies. This information is used to constrain the user in which class names and properties he/she can use in authoring rules.

The names of the classes found are collected in the *base class set* (CS_b), along with `rdfs:Resource`, the superclass of all RDF user classes. Therefore, set CS_b is constructed as follows:

$$CS_b = \{\, c \mid (c\ \texttt{rdf:type}\ \texttt{rdfs:Class}) \in RDFS \,\} \cup \{\texttt{rdfs:Resource}\}$$

where (X Y Z) represents an RDF triple and *RDFS* is the set of all triples found in the RDF Schema documents.

Except from the base class set, there also exists the *derived class set* (CS_d), which contains the names of the derived classes, i.e. the classes which lie at rule heads (*conclusions*). This set is constructed by parsing the RuleML document. 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.

$$CS_d = \{\, c \mid c \in \text{XPATH}(\texttt{//imp/\_head//atom/\_opr/rel}) \,\}$$

Here we assume that the function XPATH(*expression*) evaluates an XPath [15] expression and returns the node-set that the equivalent XPath expression would have returned. Therefore, the XPath expression `//element`, returns the set of nodes identical to `element` regardless of their position in the XML tree, the expression `./F` or `./@F` delivers the child element or attribute *F* of the current context node element. while the expression `.//F` or `.//@F` delivers all the descendant *F* elements or attributes of the current context node. Finally, the expression $./F_1/F_2$ delivers the element F2 that is a child of the element F1 that is a child of the current context node, and so on so forth. 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, that is why there is a double slash symbol between them in the above XPath expression.

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

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

BIP = {`rdf:type`, `rdfs:label`, `rdfs:comment`, `rdfs:seeAlso`, `rdfs:isDefinedBy`, `rdf:value`}

$PS_b = \{ P \mid (P$ `rdf:type rdf:Property`$) \in RDFS \} \cup BIP$

There also exists the *derived property set* (PS_d), which contains the names of the properties of the derived classes. This set is initially empty and is extended each time a new property name appears inside the `_slot` element of the `atom` in a rule head:

$PS_d = \{ P \mid P \in$ XPATH(`//imp/_head//atom/_slot/@name`)$\}$

Finally, the *full property set* (PS_f) is the union of the above two sets: $PS_f = PS_b \cup PS_d$.

4.3.1.2. Detecting Property Domains

Each of the properties in the PS_f set is further processed to detect the corresponding domains (i.e. classes). This phase also includes traversal of sub-property relations, in order to inherit super-property domains. The domain set of each property is needed, so that, for each `atom` element appearing inside the rule body, when a specific class *C* is selected, the names of the properties that can appear inside the `_slot` sub-elements are constrained only to those that have *C* as their domain. Notice that only slots of rule body atoms need to be constrained in this way, since rule heads either define entirely new classes (and slots) or they completely re-use already defined ones.

The DOM_P set of domains for each base property *P* initially contains the direct domain(s) of *P* and the inherited domains of all the (direct and indirect) superproperties of *P* (namely $SUPP_P$, which is calculated at the end of this sub-section), according to the RDFS semantics:

$$\forall P \in PS_b, DOM_P = \{c \mid (P \texttt{ rdfs:domain } c) \in RDFS\} \cup \bigcup_{x \in SUPP_P} DOM_x$$

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

$\forall P \in BIP$, $DOM_P = \{$ `rdfs:Resource` $\}$

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

$\forall P \in \{ p \mid p \in (PS_b\text{-}BIP) \wedge (\neg\exists c, (p$ `rdfs:domain` $c) \in RDFS) \}$,

$DOM_P = \{$ `rdfs:Resource` $\}$

As far as the derived properties are concerned, their domain is the derived class where they appear (in rule heads).

$\forall P \in PS_d, DOM_P = \{ c \mid$

$c \in$ XPATH(`//imp[./_head//atom/_slot/@name=`P`]/_head//atom/_opr/rel`$)\}$

Notice that inside the square brackets there is a logical expression that selects only those elements that satisfy the expression.

The *superproperty set* $SUPP_P$ of each base property P contains both the direct and indirect super-properties of P, by recursively traversing upwards the property hierarchy:

$$SUPP_P = \{ SP \mid (P \texttt{ rdfs:subPropertyOf } SP) \in RDFS \} \cup \{ SP' \mid SP' \in SUPP_{SP} \wedge SP \in SUPP_P \}$$

The properties that do not have super-properties (including the derived class properties) have an empty $SUPP_P$.

$\forall P \in PS_d, SUPP_P = \varnothing$

$\forall P \in \{ p \mid p \in PS_b \wedge (\neg\exists sp, (p$ `rdfs:subPropertyOf` $sp) \in RDFS) \}, SUPP_P = \varnothing$

4.3.1.3. Linking Classes with Properties

Since the properties are now fully described (each of them contains the corresponding super-property and domain sets), every class C in the CS_f set has to be linked with the allowed properties. More specifically, for each class C, five distinct sets have to be defined: *superclass set* $SUPC_C$, *subclass set* $SUBC_C$, *owned property set* $OWNP_C$, *inherited property set* $INHP_C$, and *subsumed property set* $SUBP_C$. *Owned properties* of a class are those properties that have this class explicitly in their domain set. *Inherited properties* of a class are those properties that have a superclass of this class in their domain. Finally, *subsumed properties* of a class are those properties that have a subclass of this class in their domain.

The $SUPC_C$ set contains all the direct and the indirect super-classes of C, by recursively traversing upwards the class hierarchy:

$$SUPC_C = \{ SC \mid (C \texttt{ rdfs:subClassOf } SC) \in RDFS \} \cup \{ SC' \mid SC' \in SUPC_{SC} \wedge SC \in SUPC_C \}$$

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

$\forall C \in CS_d, SUPC_C = \{$`rdfs:Resource`$\}$

$\forall C \in \{ c \mid c \in CS_b \wedge (\neg\exists sc, (c$ `rdfs:subClassOf` $sc) \in RDFS) \}$,

$SUPC_C = \{$`rdfs:Resource`$\}$

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

$SUBC_C = \{ sbc \mid C \in SUPC_{sbc} \}$

The $OWNP_C$ set of owned properties is constructed, by examining the domain set of each property object in the full property set:

$$OWNP_C = \{ p \mid C \in DOM_p \}$$

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

$$INHP_C = \{ p \mid p \in OWNP_{SC} \wedge SC \in SUPC_C \}$$

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

$$SUBP_C = \{ p \mid p \in OWNP_{SC} \wedge SC \in SUBC_C \}$$

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

The above mentioned three property sets comprise the *full property set* FPS_C:

$$FPS_C = OWNP_C \cup INHP_C \cup SUBP_C$$

which is used to restrict the names of properties that can appear inside a `_slot` element (see Fig. 15), when the class of the atom element is *C*, as it is explained in detail in the following sub-section.

4.3.1.4. Example

An example of all the above is shown in Table 1. Assume an RDF Schema ontology with three classes connected through a hierarchy: the class `novel` is a subclass of the `book` class and a superclass of the `graphic_novel` class. Some typical properties of these classes are displayed in the "owned properties" row. After the RDF Schema document is parsed, these classes are detected and included in the *base class set* (CS_b). Furthermore, the corresponding properties are determined and added to the *base property set* (PS_b). Eventually, every available class will be linked to the respective properties, but also to the properties of its super- and subclasses, following the rationale developed before in this section. The final status of the class properties is displayed in Table 1**.**

Table 1. Example of inherited, owned and subsumed properties

Classes / Properties	**book**	**novel**	**graphic_novel**
Owned	author title	collectible printNo	pic_artist pic_quality
Inherited		author title	author, title collectible, printNo
Subsumed	collectible, printNo pic_artist, pic_quality	pic_artist pic_quality	

This logic is reflected in the rule editor, as Fig. 15 shows. If, for example, the user wishes to formulate rule r_1 (section 2), then he/she selects the `novel` class as the value of the `href` attribute of the `_opr` element of an `atom` in the rule body and the allowed properties to be entered at the `_slot` element are all the properties included in Fig. 15. This facilitates the user, since he/she does not have to worry about which properties can be applied to `novel` instances. Notice that it is allowed to query class `novel`, demanding that property `pic_artist` satisfies a certain condition; however, only objects of class `graphic_novel` can possibly satisfy the condition, since direct instances of class `novel` do not even have `pic_artist` as a property. However, this is not a problem for the underlying inference engine of CLIPS, since the latter will transparently include only objects from class `graphic_novel` in the result.

4.3.2. RDFSbuilder – Object Modelling of RDF Schema Ontologies

The second module encompassed by the VDR-DEVICE graphical front-end is *RDFSbuilder*, a visual RDF Schema (RDFS) ontology editor [38]. RDFSbuilder emphasizes on offering a familiar development approach that closely assimilates visual object-oriented programming, i.e. using a variety of on-screen tools and drag & drop controls. However, contrary to common ontology authoring tools, the system adopts a fully object-oriented representation of the ontology model. This representation not only makes the module directly usable by users, that are not accustomed to the particular RDF Schema modelling style, which was specified as a primary design requirement of the software, but also complies with the basic design principles followed by the DR-DEVICE reasoning subsystem (see section 4.2.1).

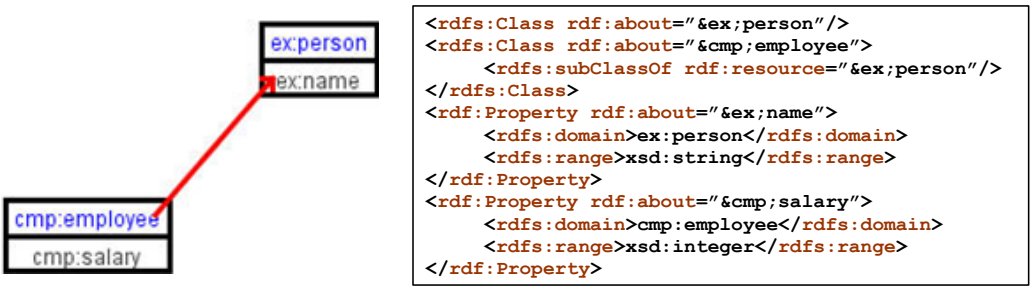


Fig. 18. Example of two classes in RDFSbuilder, accompanied by the corresponding RDF Schema fragment

According to the system design principles, properties are encapsulated as attributes in classes, resulting in a representation that appears dissimilar from the standards of the RDFS model, but is not too differentiated from UML class diagrams. Classes are represented as rectangles, properties are encapsulated by classes and the subclass relationship is represented by an arrow that commences from the subclass and ends on the superclass rectangle. Users can also define globally visible properties, by declaring `rdfs:Resource` as the property domain. Fig. 18 displays an example of two classes in RDFSbuilder that share a super-/subclass relationship, accompanied by the corresponding RDF Schema fragment.

4.3.2.1. The User Interface

Fig. 19 displays the main window of the module, which is composed of two major parts: the upper part includes the toolbar, which contains icons, representing the most common utilities of the editor, while the central part comprises the drawing panel, where the user can visually design the ontology model.

When attempting to insert a new class, the user has the option of (a) creating a completely new "empty" class, which can then be manipulated and "filled" with properties or, alternatively, (b) adding a class that belongs to an existing imported ontology. The process behind the second option is more extensively described in the following subsection.

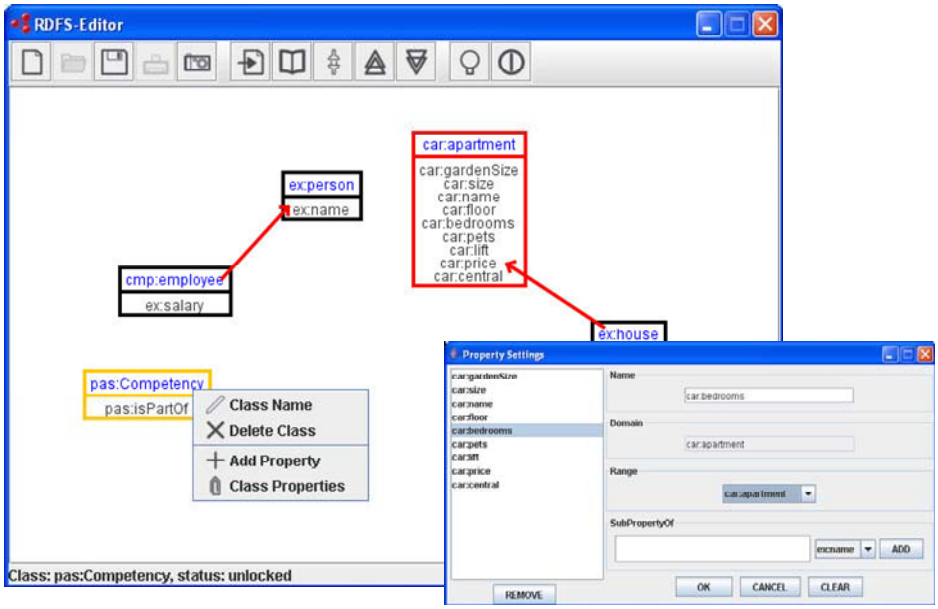


Fig. 19. The main window of the module and the properties dialog box

Naturally, the characteristics of each class can be modified, while defining subclass relationships is equally easy. The user has to indicate the subclass and the superclass, by dragging a line from the former towards the latter. A "subclass" arrow, similar to the one in Fig. 18, is then drawn that connects the two classes. Classes that have no superclass in the model are considered to be direct subclasses of `rdfs:Resource`, which represents the class of all resources.

RDFSbuilder prevents users from fundamental errors like concept hierarchy validity errors or inconsistencies in range restrictions and type inheritance, but does not handle more serious consistency errors, like cycles in hierarchies or cross-ontology inconsistencies.

4.3.2.2. Importing Classes and Properties from an Existing Ontology

Similarly to the NDW window in DRREd (see section 4.3.1), RDFSbuilder treats namespaces as addresses of input RDF Schema ontologies that contain essential vocabulary for the modeling of the ontology under development. The namespaces applied are used in pull-down menus and lists, in order to prevent potential errors on behalf of the user.

Thus, importing classes and properties from an existing RDF Schema ontology can be performed by inserting the URL of the ontology to be imported as well as a corresponding prefix (see Fig. 20). Note that a default system-defined namespace (**`ex:http://www.example.org/`**) is already declared. The module, however, does not currently handle conflicts among the imported ontologies, like disagreements in class names or inconsistencies in classes and properties. These aspects, as well as other ontology merging-related aspects, are, nevertheless included in plans for future work.

By clicking on the "Add" button, the ontology is added to the list of imported ontologies, but is not yet loaded. The user now has the choice of loading the ontology either *locked* or *unlocked*; the former allows no modifications to the classes and properties imported from the specific ontology, while the latter performs exactly the opposite functionality, allowing the user to modify the imported elements. Alternatively, users can keep the ontology in an unloaded status (i.e. ontologies simply appear in the list, but do not affect the ontology under development).

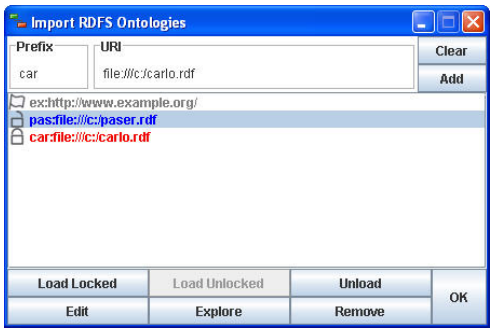


Fig. 20. Importing and RDFS ontology

The feature of locked/unlocked ontologies serves not only management purposes (e.g. include an ontology in the list for later use), but also prevents user mistakes, since a locked class, for instance, cannot be modified, but only extended. Locked ontologies are not contained in the exported RDF Schema file (only a namespace reference is included), while unlocked ontologies are fully copied into the exported file, since the system cannot automatically extend a third-party class.

When the user selects an ontology to be loaded (either locked or unlocked), the selected ontology is downloaded and parsed by *Jena Semantic Web Framework* [44], which collects all the classes and properties contained in the loaded RDFS document. The elements collected will then be available during the modelling of the ontology developed and the user is offered the capability either to insert an imported class (see previous section), including the corresponding properties (properties that have the specific class as their domain), or to apply an imported property, through the "Class Properties" menu.

Imported classes that belong to a loaded ontology can then be added to the current model, with their locked/unlocked status indicating whether they can be modified, enhanced or extended. Visually, imported classes are distinguished from the rest by their outline colour, which is *red* for the locked classes and *blue* for the unlocked.

The user can also manually "discover" more namespaces, by pressing the "*Explore*" button in the bottom of the window, similarly to NDW in DRREd (section 4.3.1). The system then downloads the namespace documents contained within the specific document and displays them in the namespaces list, accompanied by an "Unloaded" flag.

4.3.2.3. Exporting the RDF Schema Ontology

Besides RDF/XML syntax (normal and abbreviated), the RDFS ontology developed can also be exported to Notation 3 and N-Triple formats:

- *Notation 3* (N3) [16] comprises a compact and easily readable alternative to RDF's XML syntax, extended to allow greater expressiveness. N3 files typically have the extension '`.n3`'.

- *N-Triples* is a line-based, plain text format for encoding an RDF graph. It was designed to be a fixed subset of N3. N-Triples content is typically stored in files with an '`.nt`' suffix to distinguish them from N3.

4.3.3. DL-RuleViz - Visualizing a Defeasible Logic Rule Base

Realizing the need to provide graphical trace and explanation mechanisms for the derived conclusions, VDR-DEVICE is equipped with *DL-RuleViz*, a software module for visualizing defeasible logic rule bases [37]. The module's representational schema is based on directed graphs and was introduced in section 3. The current section, however, focuses on how DL-RuleViz *builds* the defeasible logic rule base digraph. Note that in our approach the visual representation of a defeasible logic rule base is bijective, meaning that every visualization implementation is mapped to by exactly one rule base.

4.3.3.1. Class Boxes, Class Patterns and Slot Patterns

In this sub-section we describe how DL-RuleViz collects the class boxes, class patterns and slot patterns to be visually drawn. To this end the input RuleML rulebase is parsed and analyzed, re-using also some sets defined in sub-section 4.3.1.1.

Class boxes are simply containers and are the equivalent of *predicate boxes* described previously. They are 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 filtered subsets of instances of the specific class. *Slot patterns* are the equivalent of *argument patterns* and *condition patterns*. However, there are 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 VDR-DEVICE 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 encapsulated inside the slot elements; 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. 21. The figure illustrates a class box that contains three class patterns applied on the *novel* class. 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 corresponding expressions and conditions, regarding the variables on the left. In the case of constant values, only the left-hand side is utilized; thus, the second class pattern of the box in Fig. 21, for example, refers to all the *collectible* novels. This way the content of the slot arguments is clearly depicted and easily comprehended. Finally, the third class pattern refers to all the novels by *Asimov* with price greater than *18€*. Fig. 22 displays a code fragment matching the third class pattern of the class box in Fig. 21, written in the RuleML-like syntax of VDR-DEVICE (v. 0.9).

To construct the class box set CB_f, for each class *c* that belongs to the base and derived full class sets (CS_b and CS_d, respectively) a *class box cb* with the same name is constructed and placed inside the corresponding *class box set* CB_b and CB_d. CB_f is the union of the latter.

$CB_b = \{ cb \mid cb \in CS_b \}$

$CB_d = \{\ cb \mid cb \in CS_d\ \}$

$CB_f = CB_b \cup CB_d$

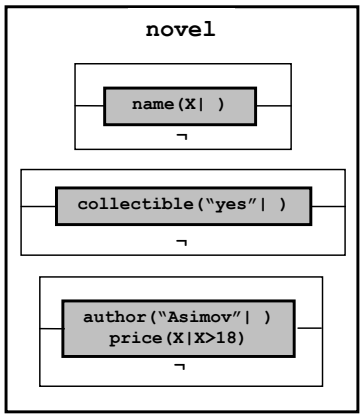


Fig. 21. A class box example that contains three class patterns.

```
<Atom>
  <op>
    <Rel uri="novel"/>
  </op>
  <slot>
    <Ind>author</Ind>
    <Data xsi:type="xs:string">Asimov</Data>
  </slot>
  <slot>
    <Ind>price</Ind>
    <Constraint>
      <and_constraint>
        <Var>x</Var>
          <Function_call name="&gt;">
          <Var>x</Var>
         <Ind>18</Ind>
        </Function_call>
      </and_constraint>
    </Constraint>
  </slot>
</Atom>
```

Fig. 22. Code fragment for the third class pattern of the class box in Fig. 21.

The class boxes are populated as follows: for each `atom` element inside a rule head or body, a new class pattern is created and inserted into (or, in a wider point of view, *associated with*) the class box, whose name 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 noticed.

$$CP_H = \{\ \langle cp,cb,r\rangle \mid r \in \text{XPATH}(\texttt{//imp}) \wedge cp \in \text{XPATH}(r\texttt{/\_head//atom}) \wedge cb \in \text{XPATH}(cp\texttt{/\_opr/rel})\ \}$$

$$CP_B = \{\ \langle cp,cb,r\rangle \mid r \in \text{XPATH}(\texttt{//imp}) \wedge cp \in \text{XPATH}(r\texttt{/\_body//atom}) \wedge cb \in \text{XPATH}(cp\texttt{/\_opr/rel/@href})\ \}$$

$CP = CP_H \cup CP_B$

$S_{cb} = \{\ cp \mid \langle cp,cb,r\rangle \in CP\ \}$

Notice that the *CP* set consists of tuples <*cp*,*cb*,*r*> where *cp* is the class pattern, *cb* is the class box and *r* is the rule where the class pattern appears in. If the class pattern

belongs to the CP_H subset, then the corresponding class pattern appears in the head of the rule, whereas if it belongs to the CP_B subset, then it appears in the body of the rule. The S_{cb} set contains the class patterns for each class box *cb*.

A class box could remain empty, in case a base class is included in the loaded RDF Schema document(s), but is not being used in any rule during the development of the rule base. However, it is obvious that this does not apply for derived classes, since the latter are dynamically detected and added to the full class set. Empty class boxes still appear in the rule graph, but naturally play a limited role.

Similarly to class boxes, class patterns are populated with one or more *slot patterns*. For each `_slot` element inside an atom, a slot pattern is created. Each slot pattern is a tuple <*sp*,*s*,*V*,*C*> that consists of the `_slot` element *sp*, the slot name s (contained inside the `name` attribute of the `_slot` element) and, a set *V* of variables and a set *C* of value constraints. The latter (*V* and *C*) are optional, i.e. the corresponding sets may be empty. The variable in the slot pattern is used in order to unify (retrieve) the slot value, with the latter having to satisfy the set of constraints. In other words, slot patterns represent conditions on slots (or class properties). Notice that if the *V* set is not empty it contains at most one element, since only one variable is allowed to retrieve the value of the corresponding slot. Each of the slot pattern "ingredients" (slot name, variables and value constraints) is being retrieved from the children (direct and indirect) of the `_slot` element in the XML tree representation of the rule base.

$$\begin{aligned} SP_{cp} = \{ <sp,s,V,C> \mid & cp \in \text{XPATH}(\texttt{//atom}) \wedge sp \in \text{XPATH}(cp/\texttt{\_slot}) \wedge \\ & s \in \text{XPATH}(sp/\texttt{@name}) \wedge \\ & V \equiv (\text{XPATH}(sp/\texttt{var}) \cup \text{XPATH}(sp/\texttt{\_and/var})) \wedge \\ & C \equiv (\text{XPATH}(sp/\texttt{ind}) \cup \text{XPATH}(sp/\texttt{\_not}) \cup \text{XPATH}(sp/\texttt{\_and/ind}) \\ & \quad \cup \text{XPATH}(sp/\texttt{\_and/function\_call})) \end{aligned}$$

Each slot pattern is associated with the class pattern *cp* of the relevant atom, through the SP_{cp} set. The union of all such sets of slot patterns is denoted by *SP*.

$$SP = \bigcup_{x \in CP} SP_x$$

Notice that the above expressions detect only the existence of a `function_call` element. Of course, in order to construct the actual constraints in the constraint set of the slot (as in Fig. 21), further drill-down of the `function_call` element is needed. However, such processing is quite a low-level issue, that we do not believe it should be discussed further.

4.3.3.2. Rule Circles and Arrow Types

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

In what follows, we first construct the five sets of arrows and then we use these sets to construct the set of rule circles. The contents of these sets are tuples that contain various pieces of information; among others they contain links from rule circles to arrows and vice-versa.

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 emanating from rule circles is denoted by *RA*. Each arrow is represented by a tuple <*r*,*cp*> between the rule *r* and the corresponding pattern *cp* of the rule head.

$$SA = \{ \langle r,cp \rangle \mid r \in \text{XPATH}(\texttt{//imp/\_rlab[@ruletype="strictrule"]/@ruleID}) \wedge \langle cp,cb,r \rangle \in CP_H \}$$

$$DA = \{ \langle r,cp \rangle \mid r \in \text{XPATH}(\texttt{//imp/\_rlab[@ruletype="defeasiblerule"]/@ruleID}) \wedge \langle cp,cb,r \rangle \in CP_H \}$$

$$FA = \{ \langle r,cp \rangle \mid r \in \text{XPATH}(\texttt{//imp/\_rlab[@ruletype="defeater"]/@ruleID}) \wedge \langle cp,cb,r \rangle \in CP_H \}$$

$$RA = SA \cup DA \cup FA$$

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

$$SRA = \{ \langle r,sr \rangle \mid r \in \text{XPATH}(\texttt{//imp/\_rlab[@superior]/@ruleID}) \wedge sr \in \text{XPATH}(\texttt{//imp/\_rlab[@ruleID=}r\texttt{]/@superior}) \}$$

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

$$CA = \{ \langle cp,r \rangle \mid \langle cp,cb,r \rangle \in CP_B \}$$

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

After the construction of arrows, the rule circles are constructed. For every rule *r* in the rule base a rule circle is constructed, which consists of the tuple <*r*,*In*,*Out*,*SupIn*,*SupOut*>, where *r* is the name of the corresponding rule, namely the value of the `ruleID` attribute in the `_rlab` element of the corresponding rule, *In* is the set of the incoming arrows from the class patterns, retrieved from the *CA* set, *Out* is the outgoing arrow from the *RA* set, *SupIn* is the set of the incoming superiority relationship arrows, retrieved from the *SRA* set, and *SupOut* is the set of the outgoing superiority relationship arrows, retrieved from the same set. The set of all rule circles is denoted by *RC*.

$$\begin{aligned} RC = \{ \langle r,In,Out,SupIn,SupOut \rangle \mid r \in \text{XPATH}(\texttt{//imp/\_rlab/@ruleID}) \wedge \\ In \equiv \{ \langle cp_{in},r,\texttt{plain} \rangle \mid \langle cp_{in},r \rangle \in CA \} \wedge \\ ((Out = \langle r,cp_{out},\texttt{plain},\texttt{strict} \rangle \wedge \langle r,cp_{out} \rangle \in SA) \vee \\ (Out = \langle r,cp_{out},\texttt{plain},\texttt{defeasible} \rangle \wedge \langle r,cp_{out} \rangle \in DA) \vee \\ (Out = \langle r,cp_{out},\texttt{plain},\texttt{defeater} \rangle \wedge \langle r,cp_{out} \rangle \in FA)) \wedge \\ SupIn \equiv \{ \langle sr,r \rangle \mid \langle sr,r \rangle \in SRA \} \wedge SupOut \equiv \{ \langle r,inf \rangle \mid \langle r,inf \rangle \in SRA \} \} \end{aligned}$$

Notice that each one of the *In* and *Out* arrows has been assigned a `plain` type; this is explained in the next subsection where the algorithm for the visual stratification of the rulebase is presented. Furthermore, the *Out* arrow has been characterized according to one of the three rule types.

4.3.3.3. The Rulebase Stratification Algorithm

After having collected all the necessary graph elements and having populated all the class boxes with the appropriate class and slot patterns (see previous sections), the next task is the placement of each element in the graph. To this end, an algorithm for the visualization of the defeasible logic rule base was implemented. At its foundations, the algorithm takes advantage of common rule stratification techniques (e.g. [61]). Unlike the latter, however, that focus on computing the minimal model of a rule set, our algorithm aims at the optimal *visualization* outcome. More specifically, the goal is to produce a highly readable graph that conveys the required information to the user, according to aesthetic criteria discussed in graph visualization literature (e.g. in [21] or [60]). These criteria are embodied in the visualization algorithm, as optimization goals. The algorithm is displayed in Fig. 23.

```
str:=1,
foreach cb∈CB do VCP_cb:=∅,
foreach cb∈CB_b do stratum_cb:=str,
foreach cb∈CB_d do stratum_cb:=MAXINT,
while |RC|≠0 do
   RuleTemp:=∅,
   str:=str+1,
   foreach <R,In,Out,SupIn,SupOut>∈RC do
      here:=true
      foreach <cp_in,R,Type>∈In do
            if (∃<cp_in,cb,R>∈CP ∧ stratum_cb>=str)
                  then here:=false
                  else if (∄<cp'_in,R,Type'>∈In ∧ ∃<cp'_in,cb',R>∈CP ∧stratum_cb'=str-1)
                        then here:=false
      if here=true
            then stratum_R:=str, RC:=RC-{<R,In,Out,SupIn,SupOut>},
                  RuleTemp:=RuleTemp∪{<R,In,Out,SupIn,SupOut>},
   foreach <R,In,Out,SupIn,SupOut>∈RuleTemp do
      In'=In,
      foreach <cp_in,R,Type>∈In do
            if (∃<cp_in,cb,R>∈CP ∧ stratum_cb=str-1)
                  then Type':=plain
                  else Type':=expandable,
            In:=In-{<cp_in,R,Type>}∪{<cp_in,R,Type'>}
            VCP_cb:= insert_cp_into_cb(cp_in,VCP_cb),
      RuleTemp:=RuleTemp-{<R,In',Out,SupIn,SupOut>}∪{<R,In,Out,SupIn,SupOut>},
   str:=str+1,
   CbTemp:=∅,
   foreach <R,In,Out,SupIn,SupOut>∈RuleTemp do
       if (∃<R,cp_out,OrientType,RType>∈Out ∧ ∃<cp_out,cb,R>∈CP_d ∧ stratum_cb=MAXINT)
       then stratum_cb:=str, CbTemp:=CbTemp∪{cb},
   foreach <R,In,Out,SupIn,SupOut>∈RuleTemp do
      if (∃<R,cp_out,OrientType,RType>∈Out ∧ ∃<cp_out,cb,R>∈CP_d ∧ cb∈CbTemp)
            then OrientType':=plain
            else OrientType':=dotted,
      Out':=Out-{<R,cp_out,OrientType,RType>}∪{<R,cp_out,OrientType',RType>},
      RuleTemp:=RuleTemp-{<R,In,Out,SupIn,SupOut>}∪{<R,In,Out',SupIn,SupOut>},
      VCP_cb:=insert_cp_into_cb(cp_out,VCP_cb)
```

Fig. 23. The rulebase stratification algorithm

As can be observed, the algorithm generates a straight-line graph, giving a left-to-right orientation to the flow of information; namely, the arcs in the digraph are directed from left to right, resulting in a less complex derived graph that minimizes crossings. 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, where each graph element (rule circle or class box) 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 and all the derived class boxes are assigned to the maximum allowed integer MAXINT.
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.

If cycles are encountered in the graph (i.e. if a conclusion of a new rule is a premise for an existing rule in a previous stratum), then neither the conclusion of the new rule is drawn again, nor the arrow connecting the new rule with the existing conclusion is drawn backwards. Instead, in order for the complexity of the graph to be reduced, 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 the class box of a class pattern and a rule circle is greater than 1 (i.e. the class box of the class pattern and the rule circle are not assigned to consecutive strata), the arrow that connects them is qualified as "*expandable*" (contrary to "*plain*"). To prevent graph cluttering, expandable arrows are not drawn by default, but can indeed be included in the graph at the user's discretion, by "*expanding*" (or *revealing*) all the arcs of the corresponding rule. Notice that there always exists at least one "plain" arrow connecting a class pattern of the previous stratum to a rule circle in the next stratum, according to the step 2a above.

Furthermore, the algorithm also features *class pattern unification*, which offers a simplified display of multiple unifiable class patterns; two or more class patterns are considered as *unifiable*, if the corresponding variables and expressions can be unified. When multiple class patterns are unifiable, only one of them is drawn. The choice among the unifiable patterns to be drawn is based on the currently focused rule; only the class patterns involved in the body and the head of the currently focused rule are displayed.

```
set of tuple function insert_cp_into_cb(CP,VCP_CB)
  foreach <RCP,ECPs>∈VCP_CB do
    if equal_class_patterns(CP,RCP) then return VCP_CB
    if equivalent_class_patterns(CP,RCP) then
      foreach ecp∈ECPs do
          if equal_class_patterns(CP,ecp) then return VCP_CB
      VCP_CB := VCP_CB - {<RCP,ECPs>} ∪ {<RCP,ECPs∪{CP}>}
      return VCP_CB
  VCP_CB := VCP_CB ∪ {<CP,∅>}
  return VCP_CB
```

Fig. 24. The main class pattern unification function.

The unification process is triggered by the `insert_cp_into_cb` function (Fig. 24) that inserts a class pattern into the set VCP_{cb}, which determines how the class patterns of a class box *cb* will be visually displayed. More specifically, this set consists of tuples <*RCP*,*ECPs*>, where *RCP* is a class pattern that will be visually displayed, whereas *ECPs* is a set of equivalent (unifiable) class patterns with the *RCP*, that will be visualized only when the rule that they are involved with is focused by the user. If

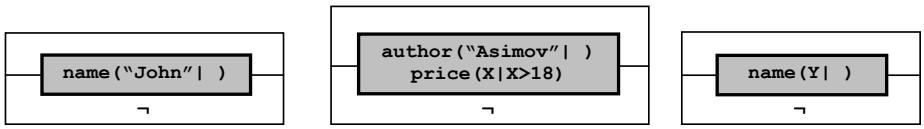
a class pattern does not have any other unifiable class patterns, then *ECPs* is empty. Notice also, that if there are equal class patterns (see below) then only one of them is ever inserted in the set either as an *RCP* or in the *ECPs* set.

In order to determine if a class pattern will be visually displayed, it is compared with every other *RCP* class pattern already included in the VCP_{cb} set and it is classified either as *equal* or *equivalent*. If it is equal, it is ignored and the VCP_{cb} set remains unchanged. If it is equivalent, then it must be added to the corresponding *ECPs* set, unless it is equal to one of the class patterns there. Finally, if nothing of the above holds, then the class pattern is added to the VCP_{cb} set.

Two class patterns are checked for equality with the function `equal_class_patterns` (APPENDIX A), by consecutively checking for equality the corresponding slot patterns (function `equal_slot_patterns`) and slot constraints (function `equal_slot_constraints`).

Two class patterns are checked for equivalence or unifiability with the function `equivalent_class_patterns` (APPENDIX A), by performing similar controls on the slot patterns (function `equivalent_slot_patterns`) and slot constraints (function `equivalent_slot_constraints`). Two patterns are unifiable either when they are equal, or when they have the same name, the same number of component patterns and each component pattern (i.e. slot, variable, constant, constraint, etc.) of one term is unifiable with one component pattern of the other term. Variables are unifiable, whereas different constants and operator symbols are not. Notice that constraints can take the following forms (this is reflected in the function `equivalent_slot_constraint`):

- *Unary constraints* "`Op Opr`", where `Op` is the operator and `Opr` is the operand, as for example "`~ 5`" (not equal to 5).
- *Binary constraints* "`Opr1 Op Opr2`", where `Op` is the operator and `Opr1`, `Opr2` are the operands, as for example "`3 | 4`" (either equal to 3 or 4).
- *Functional constraints* "`(Fun Ops)`", where `Fun` is the function name and `Ops` is the list of arguments of the function.



(a) (b) (c)

Fig. 25. Unifying class patterns (a), (b) and (c) with class patterns inserted into the class box of Fig. 21

For example, consider the class box in Fig. 21 and the three class patterns in Fig. 25. Class pattern (a) is neither equal nor equivalent to any of the existing three class patterns in the class box and is, thus, defined as *qualified for addition*, meaning that it can be safely added into the box. Class pattern (b) is *equal* to an existing class pattern (the third one) and is *not* added into the class box, while class pattern (c) is *equivalent* to an existing class pattern (the first one) and is unified with it, meaning that it is added into the class box, but only one of the two unified class patterns is displayed each time, depending on the rule being focused.

4.3.3.4. Example

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

```
r1: novel(X) → book(X)
r2: novel(X), hardcover(X) ⇒ collectible(X)
r3: novel(X) ⇒ ¬hardcover(X)
r4: novel(X),author(X,"Asimov"),price(X,Y),Y>18 ⇒ hardcover(X)
r5: novel(X),price(X,Y),
          NOT(novel(Z),Z≠X,price(Z,W),W<Y) ⇒ cheapest(X)
r6: book(X) ⇒ hardcover(X)
r4 > r3
```

Some of the rules above were encountered in a previous section, while rule r_4 reads as "*Novels by Asimov with a price greater than 18€ are typically hard-covered*" and rule r_5 reads as "*If a book with a specific price exists and there is no other book with a lower price, then the first book is considered the cheapest*". Rule r_5 is a typical example of applying negation-as-failure. Also, five classes are needed in the example, as Table 2 illustrates: one base class (`novel`) and four derived classes (`book`, `hardcover`, `collectible` and `cheapest`). Notice that, although for presentation simplicity `author` and `price` are presented as predicates in the example, in fact they are slots of the `novel` class, according to the object-oriented RDF model of the underlying system, as it was explained in section 4.2.1 and is clearly displayed in Fig. 26 and in Fig. 11.

Table 2. Classes included in the rule base of the example

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

Table 3. Stratum assignments for the classes and the rules in the example

stratum #1	`novel`
stratum #2	r_1, r_3, r_4, r_5
stratum #3	`book, hardcover, cheapest`
stratum #4	r_2, r_6
stratum #5	`collectible`

Table 3 displays the final stratum assignments, according to the algorithm. After applying the algorithm, it comes up that five strata (or columns) are needed to display all the graph elements. 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. Vertices in one column are never connected with vertices in the same column. An exception, however, is the case of rule superiorities, which is a connection type that indeed might connect rules that belong to the same stratum.

Fig. 26 displays the resulting graph, produced by DL-RuleViz, the rule base visualization module of DRREd. The implementation is totally compliant with the algorithm presented in the previous section. Notice the "dotted" arrow "leaving" rule r_6. As explained earlier, the specific 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. 26 - window on the right).

The example features a case of a "*collapsed*" rule (rule r_2), which is a rule with premises that also belong to earlier strata than the immediately previous one. In these cases, the relevant connection is not drawn by default. Instead, a "+" symbol appears

underneath the rule circle, indicating that the corresponding rule can be "*expanded*". An "*expanded*" rule then has all its "incoming" connections drawn (these connections are called "*expandable connections*", as mentioned before). Rule r_2 can be expanded, as seen in Fig. 27.

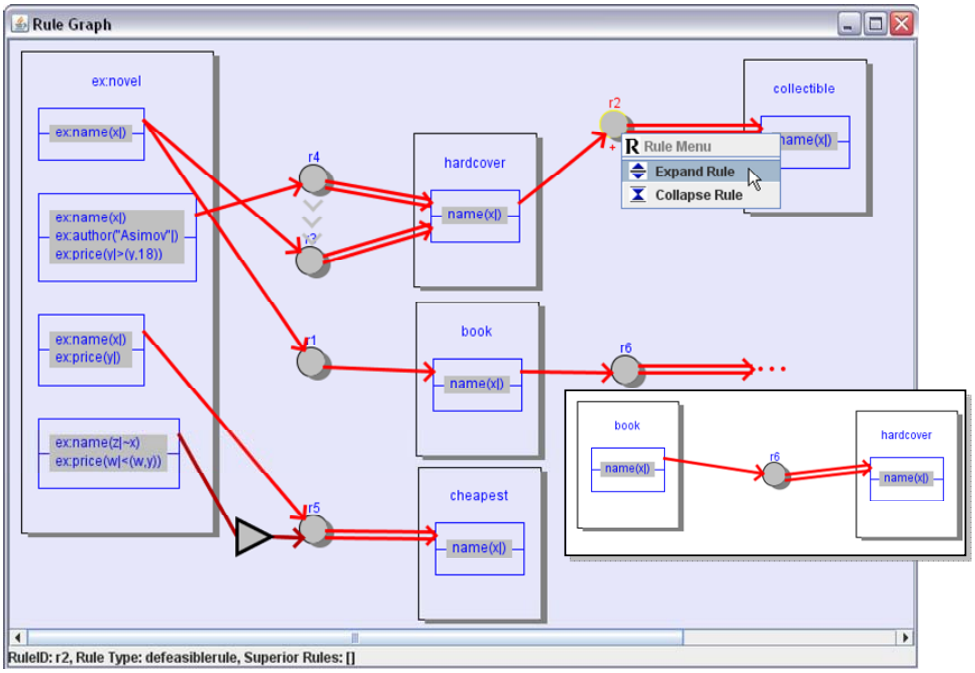


Fig. 26. Implementation of the visualization algorithm by DL-RuleViz

The derived graph also depicts the superiority relationship $r_4 > r_3$, included in the rule base, as well as negation-as-failure, exactly as the representation methodology dictates (sections 3 and 3.2). DL-RuleViz does not yet represent multi-literal truth boxes for conflicting literals, being currently under development.

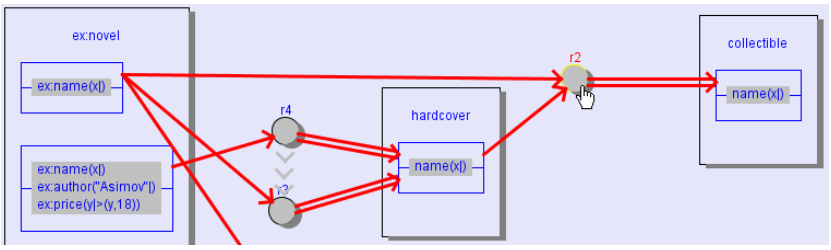


Fig. 27. A fully expanded rule (rule r_2) in DL-RuleViz

The two aspects of the rule base (the XML-tree representation, used by DRREd and the directed graph, produced by DL-RuleViz) are interrelated, meaning that traversal and alterations in one are also reflected in the other and vice versa. So, if for example the user focuses on a specific element in the tree and then switches to the digraph view, the corresponding element in the digraph is also selected and the data relevant to it displayed.

5. User Evaluation

Post-graduate students attending the Semantic Web course at our university were asked to participate in assessing two modules of VDR-DEVICE, namely DRREd and

DL-RuleViz. More specifically, they were given a defeasible logic rule base and were asked to model it, using the two components. They were also requested to answer one on-line questionnaire for each module. The questionnaire was not aimed at the usability of the software facilities; instead, its primary objective was to allow users to evaluate the representation schema adopted by each of the systems. The results are described in the following subsections.

5.1. DRREd User Evaluation

The DRREd on-line evaluation survey was divided in two major parts: the first part focused on the comprehensibility and intuitiveness of the XML-tree representation, adopted by the module, while the second part asked 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, 76% of the participants found the representation intuitive, 68% found it easily comprehensible and 52% found it interesting. On the other hand, only 40% found the interface aesthetically pleasing, 32% found it difficult to understand and 12% considered it unacceptable.

As for the latter part of the survey, 80% of the participants believed that DRREd indeed assists in the development of a defeasible logic rule base, 60% believed that the system considerably improves productivity (i.e. minimizes development time), 72% considered that the representation gives a better overview of rule dependencies, while only 28% of the users would rather use another tool.

Overall, 76% of the users were satisfied with DRREd and only 20% would prefer more features in the proposed representation. Finally, DRREd made defeasible logic attractive to 80% of the participants. The most important drawback of the system was its lack of visual representations, a task that DL-RuleViz intends to handle. The lack of help tools as well as other control mechanisms was also pointed out. All these weaknesses are planned to be dealt with in future versions of the system.

5.2. DL-RuleViz User Evaluation

Contrary to the previous one, the second on-line survey consisted of a single part, containing questions, related to the intuitiveness and user-friendliness of the proposed representation of defeasible logic rule bases. Users generally seemed to understand and appreciate the adopted representation methodology. More specifically, 72% of the participants found the representation intuitive, the same percentage considered that the representation gives a better overview of rule dependencies, 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.

Overall, the result of using the DL-RuleViz module was considered acceptable and impressive by 44% and 30% of the users, respectively. Several users would, nevertheless, prefer more features in the proposed representation (32%), while, on the other hand, DL-RuleViz 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 module include handling more than one variable in a class pattern and representing conflicting literals.

6. Related Work

There exist several systems specifically designed for the Semantic Web environment, although, to the best of our knowledge, no system exists yet that can adequately cover so many aspects of the SW architecture. *ISWIVE* [20] is, nevertheless, a paradigm of a system that attempts to integrate a variety of SW technologies. The system visualizes SW resources, using RDF and topic maps and it also provides an interactive semantic query mechanism. However, contrary to VDR-DEVICE presented in this work, ISWIVE cannot support more sophisticated types of reasoning (e.g. defeasible reasoning) and can, thus, only handle queries of limited expressiveness. VDR-DEVICE, on the other hand, is a more integrated solution, since it also offers visualization mechanisms of the rule bases developed as well as the ontologies loaded.

d-GRAPHER [48] 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 introduced the visual development of d-graphs, utilizing a representational methodology that comprised the basis of the implementation philosophy behind the rule base drawing module of VDR-DEVICE, it has, nevertheless, a couple of weaknesses: 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. The system is not able to express more "demanding" rule bases and is now considered depreciated.

Mandarax [22] is a rule platform, which provides a rule mark-up language (compatible with RuleML) for expressing rules and facts that refer to Java objects. It is based on derivation rules with negation-as-failure, top-down rule evaluation, and generating answers by logical term unification. RDF documents can be loaded into Mandarax as triplets. Furthermore, Mandarax is supported by *Oryx*, a graphical rule management tool. Oryx includes a repository for managing the vocabulary, a formal-natural-language-based rule editor and a graphical user interface library. Contrasted, the rule authoring tool of VDR-DEVICE (DRREd) lies closer to the XML nature of its rule syntax and follows a more traditional object-oriented view of the RDF data model. Furthermore, VDR-DEVICE supports both negation-as-failure and strong negation, and supports both deductive and defeasible logic rules.

Each of the following subsections outlines systems that are similar to each of the respective VDR-DEVICE subcomponents.

6.1. Defeasible Reasoning Engines

Deimos [42] is a flexible, query processing system based on Haskell. It implements several variants, but neither conflicting literals nor negation as failure in the object language. Also, the current implementation does not integrate with Semantic Web, since it is solely a defeasible logic engine (for example, there is no way to treat RDF data and RDFS/OWL ontologies; nor does it use an XML-based or RDF-based syntax for syntactic interoperability). Therefore, it is only an isolated solution, although external translation modules could provide such interoperability. Finally, it is propositional and does not support variables.

Delores [42] is another implementation, which computes all conclusions from a defeasible theory. It is very efficient, exhibiting linear computational complexity. Delores only supports ambiguity blocking propositional defeasible logic; so, it does not support ambiguity propagation, nor conflicting literals, variables and negation as

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

SweetJess [31] is another defeasible reasoning system, based on Jess. It integrates well with RuleML; however, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML-OIL like syntax of RuleML) and not on arbitrary RDF data, like DR-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms) and, although this also applies to DR-DEVICE to a large extent, the basic DR-DEVICE language [13] can support a limited form of functions in the following sense: (a) path expressions are allowed in the rule condition, which can be seen as complex functions, where allowed function names are object referencing slots; (b) aggregate and sorting functions are allowed in the conclusion of aggregate rules. Finally, DR-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules [10].

Apart from defeasible reasoning, there also exist other non-monotonic rule systems for the Semantic Web. An example is *dlvhex* [23], a reasoning engine for HEX-programs, which are non-monotonic logic programs with external and higher-order atoms. Through external atoms, HEX-programs can deal with external knowledge, such as RDF datasets or description logics bases. This functionality is similar to DR-DEVICE, which also handles RDF meta-data coming from external sources. However, the way results are returned to the user is differentiated: DR-DEVICE returns the result-objects as an RDF/XML document that includes the instances of the exported derived classes, which have been proved, while dlvhex directly prints the resultant models as output in the form of answer sets, which can be confusing for large rule sets.

6.2. Rule Editors

Besides *Oryx*, the RuleML editor described above, there exists a small number of other rule editor implementations. *RuleVISor* [43] is a rule editor paradigm, based on the Semantic Web Rule Language (SWRL), 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.

The *Protégé SWRL Rule Editor* [49] is an open-source rule editor for SWRL, developed as a plug-in to Protégé-OWL [52]. The tool allows users to switch between SWRL rule editing and editing of OWL entities, also supporting tight integration with rule engines. Similarly to DRREd, the Protégé SWRL Rule Editor performs syntactic and semantic checking, as a rule is being entered, ensuring that each rule is syntactically correct. However, the rule base design approach adopted by the editor is dissimilar from the one adopted by DRREd: the former expresses rules as sets of predicate-based logical expressions, while the latter considers rule bases as XML trees, although both the underlying rule languages (the VDR-DEVICE RuleML-like language and SWRL) are XML-based.

A third rule editor example is *WAB* [9], an axiom and rule editor, integrated in the WebODE Ontology Editor. It allows creating first order logic axioms and rules, using a graphical user interface. As expected, each rule consists of a left- and a right-hand side, with the former consisting of conjunctions of atoms and the latter consisting of a single atom. Once a rule is created, WAB checks both the syntax of the rule and its

consistency with the associated ontology and transforms the rule into Horn clauses. Nevertheless, WAB allows the display of only one rule at a time, turning the handling of voluminous rule bases into a less manageable task.

6.3. RDF Schema Ontology Editors

There exist several implementations of editors that create or manipulate RDF Schema documents. *MR3* [59] is such a tool for editing RDF-based content. It is efficient and supports all the basic functions of the RDFS model, but follows the traditional RDF visual representation, namely, the graph is created by drawing a distinct geometrical figure (i.e. ellipses and boxes) for each entity, which finally leads to a quite confusing graph. Furthermore, its utilities are somewhat complicated to use by an unfamiliar user and, thus, comprises a solution, suitable only for experienced users.

Altova SemanticWorks [1] is a commercial product, which also offers all the basic functions of the RDFS model. Users can visually design Semantic Web instance documents, vocabularies, and ontologies and then output them in either RDF/XML or N-triples formats. Its RDF representation and utilities, however, are too complicated to understand and handle, even for users familiar with RDF and RDF Schema, making the software difficult to use.

Protégé [52], is an open-source ontology editor and knowledge base framework. It supports the creation, visualization, and manipulation of ontologies not only in RDFS but also in OWL, RDF and XML Schema. Besides its advanced functionality and efficiency, Protégé also features flexible plug-in mechanisms that add extensibility to the system as well as a wide user community, involved with a variety of research and industrial projects. However, Protégé features a modeling environment, based on graphical aids and mainly a tree representation of the ontologies developed, while RDFSbuilder is purely visual. On the other hand, there also exists *OWLViz*, one of Protégé's plug-ins, but it is merely a visualization tool for OWL ontologies, not allowing development or modeling of ontologies.

Another system that represents knowledge in a graphical context is *Spectacle* [24] that facilitates the creation of information presentations that meet the needs of end users. One of Spectacle's key components is the Cluster Map, used for visualizing ontological data. Actually, the Cluster Map is used to visualize RDFS-based lightweight ontologies that describe the domain through a set of classes and their hierarchical relationships. With Spectacle, users can efficiently perform analysis, search and navigation, although the system functionality is restricted to lightweight ontologies only.

IsaViz [51] and *RDFSViz* [56] are two more similar tools. The former represents models as directed graphs using the traditional representation of a model. It is quite simple to use but with limited functionality. The latter is a web implementation of an RDF Schema visualisation service. It provides an online demo, where users can enter the URL of their own RDFS files and the corresponding graph is generated. Both systems, nevertheless, are not editors but visualization tools, similarly to OWLViz mentioned above.

6.4. Rule Base Visualization Tools

Besides d-GRAPHER, described earlier, no modern system exists yet that can visually represent defeasible logic rules. There exist, however, systems that implement rule representation and visualization. Such an example is *Strelka*, implemented as a

plug-in for the Fujaba Tool Suite [18]. Strelka is a tool for making URML models (a UML-based Rule Modelling Language) that supports modelling of derivation, production and reaction rules and serialization of URML models into the XML format R2ML. As in DL-RuleViz, a rule in Strelka is represented graphically as a circle with a rule identifier, with incoming arrows representing rule conditions or triggering events and outgoing arrows representing rule conclusions or produced actions. The system also offers visual authoring of rule bases. Its underlying visual rule representation language is quite expressive, which, however, also implies that the user has to be accustomed to its characteristics, a process that probably demands a steeper learning curve.

CViz [33] represents another attempt to visualize rule sets and, in particular, classification rules that consist of two parts: *condition part* and *decision part*. The system introduces a rather innovative visualization approach, by representing rules as strips (called *rule polygons*), which cover the area that connect the corresponding attribute values. Nevertheless, the user evaluation, performed on the system, has proved that CViz indeed assists users in understanding the relationships among data and concentrating on the meaningful data in the process of discovering knowledge. Besides the heavily differentiated representational approach, CViz's difference from DL-RuleViz can be traced in its ability to handle a vast number of classification rules, falling however short in visualizing rule chaining, something that our module aims at.

CPL (Conceptual Programming Language) [50] constitutes an effort to bridge the gap between Knowledge Representation (KR) and Programming Languages (PL). CPL is a visual language for expressing procedural knowledge explicitly as programs. The basic notion in CPL are Conceptual Graphs (CGs), which are connected, multi-labeled bipartite oriented graphs and can express declarative and/or 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 the 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.

7. Conclusions and Future Work

This paper argued that logic is currently the target of the majority of the upcoming efforts towards the realization of the Semantic Web vision and presented the basic characteristics of defeasible reasoning, which represents a rule-based approach to reasoning with incomplete and conflicting information. Defeasible reasoning is considered a potent tool in many SW-related applications. A system, based on defeasible reasoning and specifically designed for the Semantic Web environment, was also presented in this work. The system is called VDR-DEVICE[1] and it comprises a visual integrated development environment for developing and deploying defeasible logic rule bases. The system employs a user-friendly graphical shell and a defeasible reasoning system that supports direct import from the Web and processing of RDF data and RDF Schema ontologies. The graphical shell consists of a number of modelling and visualization facilities, such as graphical and visual rule base visualization modules (DL-RuleViz), a graphical rule base editor (DRREd) and an RDF Schema Ontology authoring tool (RDFSbuilder).

[1] VDR-DEVICE is available at: `http://lpis.csd.auth.gr/systems/dr-device.html`

During the development phases of VDR-DEVICE, a number of requirements and specifications were designated. The primary requirement, regarding the reasoning engine, was to support defeasible reasoning. Consequently, the rule editor and visualization modules had to adopt a suitable defeasible logic representation schema, which would prove intuitive and easy to apply. Moreover, a secondary requirement for the editors was to prevent users from syntactic and semantic errors during development. The user evaluation performed on DRREd and DL-RuleViz, also presented in this work, confirms that the above requirements are indeed met by the VDR-DEVICE modules.

Currently, we are working on potential improvements of the VDR-DEVICE system, such as: (i) enhancing DL-RuleViz with visual rule authoring utilities, (ii) adding RDF authoring capabilities to RDFSbuilder, as well as mechanisms for handling ontology conflicts, and (iii) enabling DRREd to seamlessly support newer versions of RuleML based on XML Schema. For example, an interesting potential would be to support the user through the visual rule system to take full advantage of the CLIPS capabilities behind the core reasoning module (DR-DEVICE), with e.g. built-in or user-defined functions.

On the other hand, we are also extending the core reasoning system, DR-DEVICE, to handle proofs, in order to provide proof explanation on the Semantic Web, based on defeasible reasoning. The enhancement of rule base visualization with visual rule execution tracing can ultimately lead to visualizing proofs and validating the conclusions of the system. In this way, 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.

8. References

[1] Altova, SemanticWorks. Visual Semantic Web design tool for RDF and OWL. http://www.altova.com/products/semanticworks/semantic_web_rdf_owl_editor.html, last accessed: November 20, 2006.

[2] Antoniou, G., Arief, M. Executable Declarative Business Rules and their Use in Electronic Commerce. Proc. ACM Symposium on Applied Computing, pp. 6-10, ACM Press, 2002.

[3] Antoniou, G., Billington, D., Governatori, G., Maher, M. J. A Flexible Framework for Defeasible Logics. Proc. Nat'l Conf. Artificial Intelligence (AAAI '00), pp. 405-410, 2000.

[4] Antoniou, G., Billington, D., Governatori, G., Maher, M. J. Representation results for defeasible logic. *ACM Trans. Comput. Log.* 2(2), pp. 255-287, 2001.

[5] Antoniou, G., Billington, D., Governatori, G., Maher, M. J. Embedding defeasible logic into logic programming. *Theory and Practice of Logic Programming*, 6(6), pp. 703-735, 2006.

[6] Antoniou, G., Maher, M.J., Billington, D. Defeasible Logic versus Logic Programming without Negation as Failure. *Journal of Logic Programming*, 41(1), pp. 45-57, 2000.

[7] Antoniou, G., Skylogiannis, T., Bikakis, A., Bassiliades, N. DR-BROKERING: A semantic brokering system. *Knowledge-Based Systems*, 20(1), pp. 61-72, 2007.

[8] Ashri, R., Payne, T., Marvin, D., Surridge, M., Taylor, S. Towards a Semantic Web Security Infrastructure. Proc. Semantic Web Services 2004 Spring Symposium Series, Stanford University, Stanford California, 2004.

[9] Arpírez, J.C., Corcho, O., Fernández-López, M., Gómez-Pérez, A. WebODE in a Nutshell. *AI Magazine*, 24(3):37-48, 2003.

[10] Bassiliades N., Antoniou G., Governatori G. Proof Explanation in the DR-DEVICE System. Proc. 1st International Conference on Web Reasoning and Rule Systems (RR 2007), Springer-Verlag, LNCS 4524, pp. 249-258, Innsbruck, Austria, 2007.
[11] Bassiliades N., Antoniou G., Vlahavas I. A Defeasible Logic Reasoner for the Semantic Web. *International Journal on Semantic Web and Information Systems*, 2(1), pp. 1-41, 2006.
[12] Bassiliades, N., Kontopoulos, E., Antoniou, G. A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web. Proc. RuleML-2005, pp. 172-186, Galway, Ireland, Springer-Verlag, LNCS 3791, 2005.
[13] Bassiliades N., Vlahavas I. R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata. *International Journal on Semantic Web and Information Systems*, Amit Sheth, Miltiadis D. Lytras (Ed.), Idea Group, 2(2), pp. 24-90, 2006.
[14] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P. F., Stein, L. A. OWL web ontology language reference. www.w3.org/TR/owl-ref/, W3C Recommendation, 10 February 2004.
[15] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., Simeon, J. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, W3C Recommendation, 23 January 2007.
[16] Berners-Lee, T. N3 Primer: Getting into RDF & Semantic Web using N3. http://www.w3.org/2000/10/swap/Primer, last accessed: December 19, 2007.
[17] Berners-Lee, T., Hendler, J., Lassila, O. The Semantic Web. *Scientific American*, 284(5), 34-43, 2001.
[18] Burmester S., Giese h., Niere J., Tichy M., Wadsack J., Wagner R., Wendehals L., Zündorf A. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, pp. 203-218, 2004.
[19] CLIPS Basic Programming Guide (v. 6.24). www.ghg.net/clips/CLIPS.html, last accessed: April 27, 2007.
[20] Chen, I., Fan, C., Lo, P., Kuo, L., and Yang. C. ISWIVE: An Integrated Semantic Web Interactive Visualization Environment. Proc. 19th International Conference on Advanced Information Networking and Applications (AINA'05), pp. 701-706, Volume 2, IEEE Computer Society, Washington, DC, 2005.
[21] di Battista, G., Eades, P., Tamassia, R., Tollis, I.G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
[22] Dietrich, J., Kozlenkov, A., Schroeder, M., Wagner, G. Rule-based Agents for the Semantic Web. *Electronic Commerce Research and Applications*, 2(4), pp. 323–338, 2003.
[23] Eiter T., Ianni G., Schindlauer R., Tompits H. dlvhex: A System for Integrating Multiple Semantics in an Answer-Set Programming Framework. Proc 20th Workshop on Logic Programming and Constraint Systems (WLP '06), M. Fink, H. Tompits, and S. Woltran (Ed.), pp. 206-210, TU Wien, Inst. f. Informationssysteme, TR 1843-06-02, 2006.
[24] Fluit, C., Sabou, M., van Harmelen, F. Ontology-Based Information Visualization. *Visualizing the Semantic Web*, Springer-Verlag, pp. 36-48, 2003.
[25] Gallo, G., Longo, G., Pallottino, S. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2), pp. 177-201, 1993.
[26] Gottlob, G. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation*, 2, pp. 397-425, 1992.
[27] Governatori, G. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3), pp. 181-216, 2005.
[28] Governatori, G., Dumas, M., Hofstede, A. ter, Oaks P. A formal approach to protocols and strategies for (legal) negotiation. Proc. 8th International Conference of Artificial Intelligence and Law, pp. 168-177, ACM Press, 2001.
[29] Governatori, G., Maher, M. J., Antoniou, G., Billington, D. Argumentation Semantics for Defeasible Logic. *Journal of Logic and Computation*, 14(5), pp.675-702, 2004.

[30] Grosof, B. N. Prioritized conflict handing for logic programs. Proc. 1997 Int. Symposium on Logic Programming, pp. 197-211, 1997.
[31] Grosof, B. N., Gandhe, M.D., Finin, T.W. SweetJess: Translating DAMLRuleML to JESS. Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web. Held at 1st Int. Semantic Web Conference, 2002.
[32] Grosof, B. N., Poon T.C. SweetDeal: Representing Agent Contracts with Exceptions Using Semantic Web Rules, Ontologies, and Process Descriptions. *International Journal of Electronic Commerce (IJEC)*, Special Issue on Web E-commerce, 8(4):61-98, 2004.
[33] Han, J., An, A., Cercone, N. CViz: An Interactive Visualization System for Rule Induction. Proc. 13th Biennial Conference of the Canadian Society on Computational Studies of intelligence: Advances in Artificial intelligence, H. J. Hamilton (Ed.), pp. 214-226, LNCS, vol. 1822. Springer-Verlag, London, 2000.
[34] Harary F. *Graph Theory*. Addison-Wesley, Reading, MA, 1994.
[35] Huang, Z., van Harmelen F., ten Teije, A. Reasoning with inconsistent ontologies: Framework, Prototype, and Experiment. *Semantic Web Technologies: Trends and Research in Ontology-based Systems*, John Davies, Rudi Studer, Paul Warren (Ed.), pp. 71-93, John Wiley and Sons, Ltd., 2006.
[36] Kautz, H. A., Selman, B. Hard Problems for Simple Default Theories. *Artificial Intelligence*, 28, pp. 243-279, 1991.
[37] Kontopoulos, E., Bassiliades, N., Antoniou, G. Visualizing Defeasible Logic Rules for the Semantic Web. Proc. 1st Asian Semantic Web Conference (ASWC'06), pp. 278-292, Beijing, China, Springer-Verlag, LNCS 4185, 2006.
[38] Kontopoulos, E., Kravari, K., Bassiliades, N. Object-Oriented Modelling of RDF Schema Ontologies. Proc. 11th Pan-Hellenic Conference on Informatics (PCI 2007), pp. 479-489, Patras, Greece, 18-20 May 2007.
[39] Maher M. J. A model-theoretic semantics for defeasible logic. Proc. Workshop on Paraconsistent Computational Logic, pp. 67-80, 2002.
[40] Maher M. J. Propositional Defeasible Logic has Linear Complexity. *Theory and Practice of Logic Programming*, 1(6), pp. 691-711, 2001.
[41] Maher M. J., Governatori G. A Semantic Decomposition of Defeasible Logics. Proc. AAAI'99, pp. 299-305, 1999.
[42] Maher, M. J., Rock, A., Antoniou, G., Billington, D., Miller, T. Efficient Defeasible Reasoning Systems. *Int. Journal of Tools with Artificial Intelligence*, 10(4), pp. 483-501, 2001.
[43] Matheus, C., Kokar, M., Baclawski, K., Letkowski, J. An Application of Semantic Web Technologies to Situation Awareness. Proc. 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland, 2005.
[44] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. Proc. 2nd Int. Workshop on the Semantic Web, 2001.
[45] Nute, D. A Decidable Quantified Defeasible Logic. D. Prawitz, B. Skyrms, D. Westerståhl (Ed.), *Logic, Methodology and Philosophy of Science IX*, Elsevier Science B.V, pp. 263–284, 1994.
[46] Nute, D. Defeasible Reasoning. Proc. 20th Int. Conference on Systems Science, pp. 470-477, IEEE Press, 1987.
[47] Nute, D., Erk, K. Defeasible logic graphs: I. Theory. *Decis. Support Syst.*, 22(3), pp. 277-293, 1998.
[48] Nute, D., Hunter, Z., Henderson, C. Defeasible logic graphs: II. Implementation. *Decis. Support Syst.*, 22(3), 295-306, 1998.
[49] O'Connor, M. J., Knublauch, H., Tu, S.W., Grossof, B., Dean, M., Grosso, W.E., Musen, M.A. Supporting Rule System Interoperability on the Semantic Web with SWRL. Proc. 4th International Semantic Web Conference (ISWC), Galway, Ireland, 2005.

[50] Pfeiffer, H. D., Hartley, R. T. Temporal, Spatial, and Constraint Handling in the Conceptual Programming Environment, CP. *Journal for Experimental and Theoretical AI*, 4:2, pp.167-182, 1992.
[51] Pietriga, E. IsaViz: A Visual Authoring Tool for RDF. http://www.w3.org/2001/11/IsaViz/, last accessed: November 20, 2006.
[52] Protégé Ontology Editor and Knowledge Acquisition System. http://protege.stanford.edu/, last accessed: December 4, 2006.
[53] Reiter, R. A logic for default reasoning. *Artificial Intelligence Journal,* 13, pp. 81-132, 1980.
[54] Rissland, E.L., Skalak, D.B. CABARET: rule interpretation in a hybrid architecture. *Int. J. Man-Mach. Stud*. 34(6), pp. 839–887, 1991.
[55] Schild, U. J., Herzog, S. The use of meta-rules in rule based legal computer systems. Proc. 4th Int. Conf. on Artificial Intelligence and Law (ICAIL'93), pp. 100-109, ACM Press, 1993.
[56] Sintek, M., Lauer, A. The FRODO RDFSViz Tool. http://www.dfki.uni-kl.de/frodo/RDFSViz/, last accessed: November 20, 2006.
[57] Skylogiannis T., Antoniou G., Bassiliades N., Governatori G., Bikakis A. DR-NEGOTIATE – A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies. *Data & Knowledge Engineering*, Elsevier, 63(2), pp. 362-380, 2007.
[58] Stephens, S. The Enterprise Semantic Web: Technologies and Applications for the Real World. *The Semantic Web: Real-World Applications from Industry*, J. Cardoso, M. Hepp, M. Lytras (Ed.), pp. 17-37, http://www.springerlink.com/content/vm612k7207406570, Springer-Verlag, to be published 2008.
[59] Takeshi, M., Noriaki, I., Naoki, F., Takahira, Y. A Graphical RDF-based Meta-Model Management Tool. *IEICE Transactions on Information and Systems*, Special Issue on Knowledge-Based Software Engineering, Vol.E89-D, No.4, pp 1368-1377, 2006.
[60] Tamassia, R. Graph drawing. *Handbook of Discrete and Computational Geometry*. J. E. Goodman and J. O'Rourke (Ed.), pp. 815-832, CRC Press, 1997.
[61] Ullman, J. D. Principles of Database and Knowledge-Base Systems. Vol. 1, *Computer Science Press*, 1988.
[62] Wagner G. Web Rules Need Two Kinds of Negation. Proc. 1st Workshop on Semantic Web Reasoning, pp. 33-50, LNCS 2901, Springer, 2003.

9. APPENDIX A

In the following we present the functions for checking if two class patterns are either equal or equivalent, which is needed in order to determine if a class pattern needs to be visualized separately in a class box or it can be unified with another one (section 4.3.3.3). Notice that the "`foreach x∈X`" construct denotes an iteration of the variable **x** over all members of the set **X**, and has nothing to do with the universal quantifier of first-order logic.

```
boolean function equal_class_patterns(cp1,cp2)
  SP1:=SP_cp1
  SP2:=SP_cp2
  foreach sp1∈SP1 do
    SP2_before:=SP2,
    foreach sp2∈SP2 do
      if equal_slot_patterns(sp1,sp2) then SP2:=SP2-{sp2}, break,
    if SP2_before=SP2 then return FALSE
  if SP2=∅ then return TRUE else return FALSE

boolean function equal_slot_patterns(<sp1,s1,V1,C1>,<sp2,s2,V2,C2>)
  if s1≠s2 then return FALSE
  if V1≠V2 then return FALSE
```

```
  if NOT equal_slot_constraints(C1,C2) then return FALSE
  return TRUE

boolean function equal_slot_constraints(cs1,cs2)
  foreach c∈cs1 do
    if c∈cs2 then cs2:=cs2-{c} else return FALSE
  if cs2=∅ then return TRUE else return FALSE

boolean function equivalent_class_patterns(cp1,cp2)
  SP1:=SP_cp1
  SP2:=SP_cp2
  foreach sp1∈SP1 do
    SP2_before:=SP2,
    foreach sp2∈SP2 do
      if equivalent_slot_patterns(sp1,sp2) then SP2:=SP2-{sp2}, break,
    if SP2_before=SP2 then return FALSE
  if SP2=∅ then return TRUE else return FALSE

boolean function equivalent_slot_patterns(<sp1,s1,V1,C1>,<sp2,s2,V2,C2>)
  if s1≠s2 then return FALSE
  if NOT equivalent_slot_constraints(C1,C2) then return FALSE
  return TRUE

boolean function equivalent_slot_constraints(cs1,cs2)
  foreach c1∈cs1 do
    cs2_before:=cs2,
    foreach c2∈cs2 do
      if equivalent_slot_constraint(c1,c2) then cs2:=cs2-{c2}, break,
    if cs2_before=cs2 then return FALSE
  if cs2=∅ then return TRUE else return FALSE

boolean function equivalent_slot_constraint(c1,c2)
  if c1=c2 then return TRUE
  if c1="Op Opr1A" ∧ c2="Op Opr2A" ∧ is_var(Opr1A) ∧ is_var(Opr2A) then return TRUE
  if c1="Opr1A Op Opr1B" ∧ c2="Opr2A Op Opr2B" ∧ unifiable(Opr1A,Opr2A) ∧
      unifiable(Opr1B,Opr2B)
    then return TRUE
  if c1="(function Ops1)" ∧ c2="(function Op2)" ∧ unifiables(Operands1,Operands2)
    then return TRUE
  return FALSE

boolean function unifiables(As,Bs)
  while As≠∅ do
    if NOT unifiable(first(As),first(Bs)) then return FALSE,
    As:=rest(As),
    Bs:=rest(Bs)
  if Bs=∅ then return TRUE else return FALSE

boolean function unifiable(A,B)
  if A=B then return TRUE
  if is_var(A) ∧ is_var(B) then return TRUE
  return FALSE
```