Visual Development of Defeasible Logic Rules for the Semantic Web

Efstratios Kontopoulos[1], Nick Bassiliades[2], Grigoris Antoniou[3]

[1]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
Tel: ++302310998231, Fax: ++302310998418
skontopo@csd.auth.gr

[2]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
Tel: ++302310997913, Fax: ++302310998419
nbassili@csd.auth.gr

[3]Institute of Computer Science, FO.R.T.H.
P.O. Box 1385, GR-71110, Heraklion, Greece
Tel: ++30 2810 391624, Fax: ++30 2810 391638
antoniou@ics.forth.gr

Visual Development of Defeasible Logic Rules for the Semantic Web

Abstract
This chapter is concerned with the visualization of defeasible logic rules in the Semantic Web domain. Logic plays an important role in the development of the Semantic Web and defeasible reasoning seems to be a very suitable tool. However, it is too complex for an end-user, who often needs graphical trace and explanation mechanisms for the derived conclusions. Directed graphs can assist in this affair, by offering the notion of direction that appears to be extremely applicable for the representation of rule attacks and superiorities in defeasible reasoning. Their applicability, however, is balanced by the fact that it is difficult to associate data of a variety of types with the nodes and the connections between the nodes in the graph. In this chapter we try to utilize digraphs in the graphical representation of defeasible rules, by exploiting the expressiveness and comprehensibility they offer, but also trying to leverage their major disadvantages. Finally, the chapter briefly presents a tool that implements this representation methodology.

Keywords: Non-monotonic Reasoning, Defeasible Logic, Semantic Web, Visual Representation, Information Visualization, Inference Systems

INTRODUCTION

The Semantic Web (Berners-Lee, 2001) 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. The development of the Semantic Web proceeds in layers, each layer being on top of other layers. This architecture imposes the substantial evolvement of the previous layers, before a specific layer can commence getting developed. The upcoming efforts towards the development of the Semantic Web will be targeted at the Logic and Proof layers, which are believed to posses a key role in the eventual acceptance of the Semantic Web on behalf of the users.

Defeasible reasoning (Nute, 1987), a member of the non-monotonic reasoning family, represents a simple rule-based approach to reasoning not only with incomplete or changing but also with conflicting information. Defeasible reasoning can represent facts, rules and priorities and conflicts among rules. Nevertheless, defeasible logic is based on solid mathematical formulations and is, thus, not fully comprehensible by end users, who often need graphical trace and explanation mechanisms for the derived conclusions.

Directed graphs can assist in confronting this drawback. They are a powerful and flexible tool of information visualization, offering a convenient and comprehensible way of representing relationships between entities (Diestel, 2000). Their applicability, however, is balanced by the fact that it is difficult to associate data of a variety of types with the nodes and the connections between the nodes in the graph.

In this chapter we try to utilize digraphs in the graphical representation of defeasible logic rules, by exploiting the expressiveness and comprehensibility they offer, but also trying to leverage their major disadvantage, by proposing a representation approach that features two distinct node types, for rules and atomic formulas, and four distinct connection types for each rule type in defeasible logic and for superiority relationships. The chapter also briefly presents a tool that implements this representation methodology.

The rest of the chapter is organized as follows: The next section presents the vision of the Semantic Web, reviewing the most important technologies for its development. Then the importance of Logic in the Semantic Web domain is analyzed. The following two sections describe the key aspects of our approach, namely, the application of directed graphs in the representation of logic rules in general and of defeasible logic rules in particular. Then a system that implements this approach, called VDR-DEVICE, is presented and the chapter ends with the conclusions and poses future research directions.

THE VISION OF THE SEMANTIC WEB

One of the most inspired definitions for the Semantic Web (SW) was given by the inventor of the current Web - and visionary of the SW - Tim Berners-Lee (2001), according to which "*the Semantic Web is an extension of the current Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation*" (p. 35). In other words, the Semantic Web is not a mere substitute but an important enhancement to the World Wide Web (WWW) and its primary goal is to make the content of the Web accessible not only to humans, as it is today, but to machines as well. This way, software agents, for example, will have the capability to "*understand*" the meaning of the information available on the WWW and this will result in a better level of 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 efficiently by a variety of distinct Web applications.
However, the SW has not yet acquired a concrete substance and still remains a vision to a great extend. Nevertheless, a significant number of researchers and companies are working towards this direction, developing technologies that will assist the SW initiative as well as applications specifically suitable for the SW environment. These efforts are also backed by generous funding from *DARPA* (Defense Advanced Research Projects Agency of the United States) and from the *EU* (European Union).
The development of the SW is based on technologies that form a hierarchy of layers; each layer is based on the layers and technologies below it. Every layer has to be widely approved by users, companies and trust groups, before the next technologies in the hierarchy can be further developed. One of the basic layers in this hierarchy deals with content representation, where *XML* (*eXtensible Markup Language*) is the dominant standard, allowing the representation of structured documents, using custom-defined vocabulary. However, although XML offers the capability of representing virtually any kind of information, it does not offer any further knowledge, regarding the semantic meaning of the information described. Thus, it is not clear to the user what the nesting of tags, for example, means for each application. The need to represent meaning along with the content led to *RDF* (*Resource Description Framework*), a statement model with XML syntax. RDF allows representation of data and *meta-data*, a term that refers to "data about data". Meta-data are important for the SW initiative, since they also capture the meaning (or *semantics*) of the data. RDF does not supply, however, tools for structuring and organizing the data. The need for meta-data vocabularies was imperative and this was the role played by the *ontologies*. Ontology languages take representation a step further, by allowing expression of high-level semantics. *RDF Schema* and *OWL* (*Web Ontology Language*) are the main ontology-building tools in the SW. Furthermore, the *Logic* layer, one of the top layers in the SW technologies hierarchy, is based on the previous layers and offers the representational capabilities of predicate logic as well as the possibility of correlating the data. The Logic layer is accompanied by the *Proof* layer that describes the inference mechanisms, which utilize the knowledge produced from the former layer. Finally, the *Trust* layer offers trust mechanisms that allow the verification of the data exchanged in the SW.
The last layers of the SW hierarchy (i.e. the Logic, Proof and Trust layers) are not yet fully developed and, naturally, there exist almost no tools that perform such functionalities. Future attempts in the development of the SW will, nevertheless, be directed towards the evolution of these layers as well, with a number of logic-based systems emerging to bridge the gap.

RDF – A Common Information Exchange Model in the Semantic Web

An important step towards augmenting the available information in the Web with semantic content involves the design and implementation of a common information exchange model that will be used by all the SW applications. The solution, as stated in the previous section as well, lies in the *Resource Description Framework* (*RDF*), a generic model "*for supporting resource description, or metadata (data about data), for the Web*" (Powers, 2003, p. 20).
The basic idea behind RDF is a model, comprised of a number of *statements* and connections between these statements. Thus, the statement is the basic building block of an RDF document and it consists of a resource-property-value triple:

- *Resources* are the objects we want to refer to or talk about. Every resource is uniquely identified by a Uniform Resource Identifier (URI).

- *Properties* are used to describe relations between resources, but, in practice, they are a special kind of resources, being also identified by URIs.
- *Values* can either be resources or simply literals (strings).

An example of a statement that better illustrates the above is:

The apartment `http://www.example.org/carlo_ex.rdf#ap01` has a property `http://www.example.org/carlo.rdf#pets` whose value is "`yes`".

This statement declares that pets are indeed allowed in a specific apartment with codename "`ap01`". Here "`http://www.example.org/carlo_ex.rdf#ap01`" is the resource (or *subject*), "`http://http://www.example.org/carlo.rdf#pets`" is the property (or *predicate*) and the value (or *object*) is "`yes`".

There are three ways to represent an RDF statement: The first one is also the simplest one and has to do with creating a subject-predicate-object triple from the statement. Thus, the statement above would be represented by the triple:
(`http://www.example.org/carlo_ex.rdf#ap01`, `http://www.example.org/carlo.rdf#pets`, "`yes`"). Note here that the resource and property are uniquely identified by a URL (which is a kind of URI) and the value is simply a string (but could as well be a URI).

The second way of representing an RDF statement is graph-based, as can be seen in Fig. 1. A directed graph is utilized, with arcs directed from the resource to the value. Such graphs are more often encountered with the name of *semantic nets* in the AI domain.

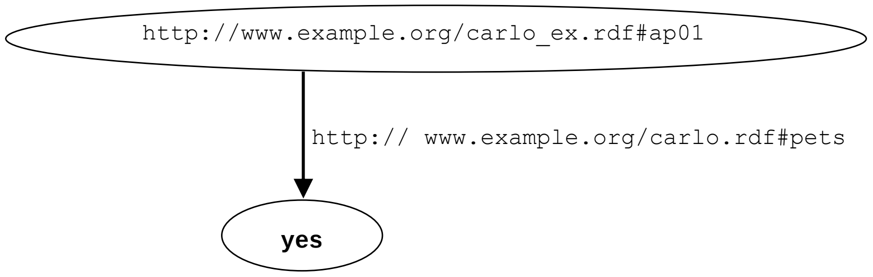

http://www.example.org/carlo_ex.rdf#ap01
http:// www.example.org/carlo.rdf#pets
yes

Fig. 1. Graph representation of an RDF statement

The third representation possibility, however, is probably the most important one, since it is based on XML and, thus, allows the re-usability of the various tools available for XML processing (*syntactic interoperability*). The XML fragment that expresses the above statement can be seen in Fig. 2.

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:carlo="http://www.example.org/carlo.rdf"
 xmlns:carlo_ex ="http://www.example.org/carlo_ex.rdf">
      <rdf:Description rdf:about="&carlo_ex;ap01">
             <carlo:pets> yes </carlo:pets>
      </rdf:Description>
<rdf:RDF>
```

Fig. 2. XML-based syntax of an RDF statement

Every statement is represented by an `rdf:Description` element. The subject of the statement is being referred to in the `rdf:about` attribute, the predicate is used as a tag and the object is the content of the tag.

Finally, the important aspect of *namespaces* should be pointed out. Namespaces define a mechanism for resolving name clashes if more than one document is imported. According to namespaces, the names (actually URIs) of the elements in an XML/RDF document are defined using a combination of a base URL, using a prefix that is declared with the command: `xmlns:prefix="URL"`, and a local name that is unique within the base URL. This way the same local name can be used freely in many RDF/RDF Schema documents, since the existence of the prefix disambiguates things. The namespaces defined in an element can be used by that element and its descendants.
In RDF, external namespaces do not only offer disambiguation, as in XML, but they are also expected to be RDF documents defining resources, which are then used in the importing RDF document. This mechanism allows the reuse of resources by other people who may decide to insert additional features into these resources. The result is the emergence of large, distributed collections of knowledge.
In the example above (Fig. 2), three namespaces are declared: `rdf`, `carlo` and `carlo_ex`. The first one contains the necessary vocabulary for RDF statements, while the other two include vocabulary specific for the domain of apartment renting.

Ontologies in the Semantic Web

The main knowledge representation tool in the SW is the *ontology*, which is simply a structured representational formalism that shares a lot of common elements with the *frames* and the *semantic nets*. The ontologies are mainly used in defining common vocabularies, used in the exchange of information among SW applications.
However, ontologies also offer interoperability among the available information and the various WWW applications like search engines, Web portals, intelligent agents and Web services. For example, the major disadvantages of today's search engines are their low precision and high recall. If there existed the possibility to perform a search based on an ontology, then the input keywords would be associated with the intended meaning. The search results would not only be more accurate, but a number of results that would contain conceptual synonyms of the inserted keyword would also be returned.
The main ontology languages in the SW today, as mentioned earlier in this chapter, are RDF Schema and OWL. The former is a vocabulary description language for describing properties and classes of RDF resources, while the latter is a richer vocabulary description language that can also describe relations between classes, cardinality, equality etc. The system VDR-DEVICE, described later in this chapter, utilizes the RDF Schema technology.

RDF Schema

Although RDF lets the user describe resources using a custom vocabulary, apparently the level of expressivity it offers is not sufficient. RDF Schema, RDF's "*semantic extension*", equips users with mechanisms for describing domains and correlations among resources, namely the allowed vocabulary for resources and properties, i.e. resources types and property types.
Though there are significant differences between RDF Schema and conventional object-oriented programming languages, the former is also based on classes and properties, similarly to the latter. Therefore, *classes* are sets of resources, while members of classes are called *instances*. On the other hand, users can also describe specific *properties* of class instances; properties impose a relation between subject and object resources and are characterized by their *domain* (the class of

resources that may appear as subjects in a triple with the property as predicate) and *range* (the class of resources that may appear as values in a triple with the property as predicate).
Instances are associated to the corresponding classes by declaring their type, using a statement like: `carlo_ex:ap01 rdf:type carlo:apartment`. This statement declares that instance `ap01` belongs to the class `apartment`. However, this is indirectly extracted by the fact that `rdf:type` has `rdfs:Class` as range, which immediately results in `apartment` being a class, according to the descriptional nature of the RDF semantics.
Naturally, the notions of hierarchy and inheritance are also applied in RDF Schema. However, they are not only applied in the case of classes but in the case of properties as well. Thus, there can be defined *super*- and *subclasses* of a specific class, but there can also exist *super*- and *subproperties* of a specific property.
An example of all the above can be seen in Fig. 3. Two classes are described: class "`house`" and its subclass "`apartment`". Similarly, two properties are also described: "`size`", which has class "`house`" as its domain and "`gardenSize`", which is a subproperty of "`size`" and, thus, inherits from the latter the domain and range. Note here that, contrary to object-oriented programming, properties are not part of a class description, but are "*globally*" visible in an RDF Schema ontology. This not only imposes a different programming approach, but also offers flexibility in extending ontologies.

```
<rdfs:Class rdf:ID="house"/>

<rdfs:Class rdf:ID="apartment">
      <rdfs:subClassOf rdf:resource="#house"/>
</rdfs:Class>

<rdf:Property rdf:ID="size">
      <rdfs:domain rdf:resource="#house"/>
      <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>

<rdf:Property rdf:ID="gardenSize">
      <rdfs:subPropertyOf rdf:resource="#size"/>
</rdf:Property>
```

Fig. 3. Classes and Properties in RDF Schema

LOGIC IN THE SEMANTIC WEB

As stated in a previous section, the *Logic* and *Proof* layers of the Semantic Web architecture constitute the current targets of most development efforts towards the SW vision. Contrary to ontologies, logic can capture more complex semantic relationships between metadata and allow processing of the latter in various useful ways.
Rules usually arise naturally from the business logic of applications. Uses of logic and/or rules include the following:

- Logic (predicate logic) can capture some of the ontological knowledge expressed in description logics (e.g. OWL). For example, the fact that class **lecturer** is a subclass of **faculty** is represented as: **lecturer(X) → faculty(X)**
- Rules can be used to convert data and/or metadata documents from one format or type to another.

- Logic can be used to check the consistency of an ontology or a metadata document using the declarations and constraints of the ontology.
- Rules can be used to formulate questions in order to retrieve resources that belong to specified metadata terms or even to identify-classify unknown terms.
- Finally, intelligent Semantic Web agents can use logical rules for decision support and action selection among a number of possibilities, in a non-deterministic way.

One of the most important advantages of using logic is that it is easy for an agent to provide *explanations* to the user about its conclusions. Simply, the explanations are formed by the sequence of inference steps that the agent's inference mechanism has followed. Explanations are very important for the adoption of Semantic Web, because they increase the confidence of users to agents' decisions.

Non-monotonic And Defeasible Reasoning

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

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

Defeasible reasoning can represent facts, rules and priorities and conflicts among rules. Such conflicts arise, among others, from rules with exceptions, which are a natural representation for policies and business rules (Antoniou et al, 1999) and priority information is often available to resolve conflicts among rules.

Conflicting Rules In The Semantic Web

Conflicting rules might be applied in the Semantic Web in the following cases:

Reasoning with Incomplete Information

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

Rules with Exceptions

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

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

Ontology Merging
When ontologies from different authors and/or sources are merged, contradictions arise naturally. Predicate logic based formalisms, including all current Semantic Web languages, cannot cope with inconsistencies. If rule-based ontology languages are used (e.g. DLP (Grosof et al, 2002)) and if rules are interpreted as defeasible (that is, they may be prevented from being applied even if they can fire) then we arrive at non-monotonic rule systems. A skeptical approach, as adopted by defeasible reasoning, is sensible because it does not allow for contradictory conclusions to be drawn. Moreover, priorities may be used to resolve some conflicts among rules, based on knowledge about the reliability of sources or on user input). Thus, non-monotonic rule systems can support ontology integration.

LOGIC AND VISUAL REPRESENTATION

There exists a variety of systems that implement rule representation and visualization, although, to the best of our knowledge, no system exists yet that can visually represent defeasible logic rules.

WIN-PROLOG

WIN-PROLOG (Steel, 2005) is a well-known Prolog compiler system, developed by LPA (Logic Programming Associates). It offers rule representation in the form of graphs as well as rule execution tracing (Fig. 4). The graphs produced, however, feature an elementary level of detail and, therefore, do not assist significantly in the visualization of the rule bases developed.

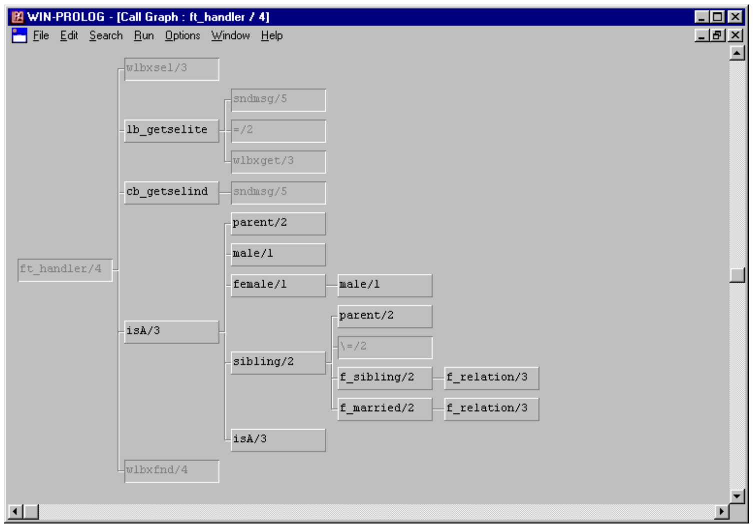


Fig. 4. Explanation trace, produced by WIN-PROLOG

VisiRule

LPA is also the developer of *VisiRule*, a graphical tool for delivering business rule and decision support applications (Shalfield, 2005). All the user has to do is draw a flowchart that represents the decision logic and VisiRule will produce Flex code from the flowchart and compile it. The system offers guidance during the construction process, constraining errors, based on the semantic content of the emerging program. This reduces the potential for constructing invalid or meaningless links, improving productivity and helping detect errors as early as possible within the design process (Fig. 5).

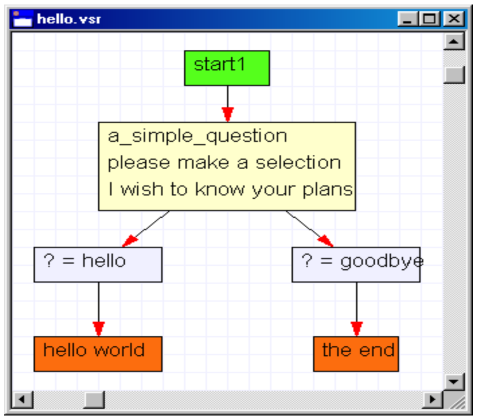


Fig. 5. Rule graph, produced by VisiRule

KEE

Certain knowledge-based system development tools also feature rule and execution graph-drawing. An example is *KEE* (Knowledge Engineering Environment) (Intellicorp, 1984) that offers several execution control mechanisms. The main features of the software include: (a) a knowledge base development tool, (b) utilities for the interface with the user and (c) graph drawing tools for the knowledge base and execution. In this case, however, the visualizations offer little or no interaction with the user and simply offer an alternative view of the knowledge base (Fig. 6).

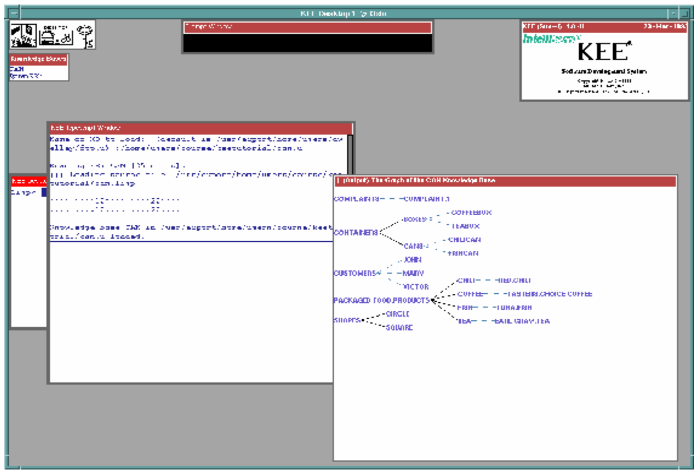


Fig. 6. Screenshot of KEE, illustrating the various tools of the system

Graphviz

Another example is *Graphviz*, an open-source graph visualization software with several main graph layout programs. Its applications are not limited to drawing rule graphs, but can also include other domains, like software engineering, database and web design, networking and visual interfaces. As a general-purpose graph drawing utility, Graphviz can be applied in rule graph drawing, since it offers variation in graph node types, but does not feature variety in the connection types in the graph and is, therefore, unsuitable for the representation of defeasible rules (Fig. 7).

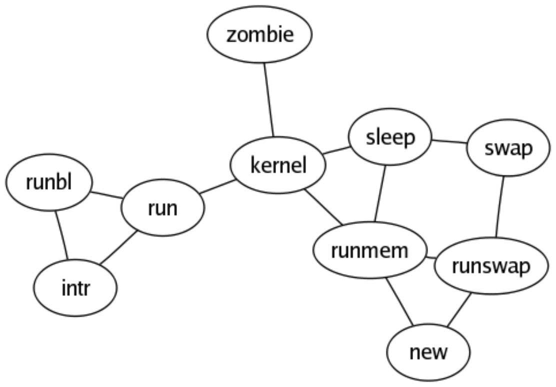
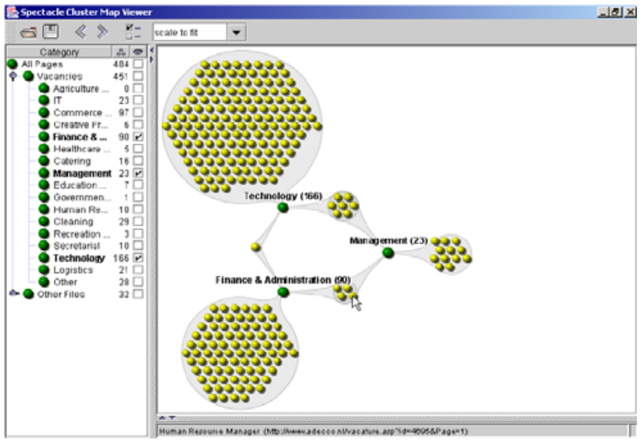


Fig. 7. A simple graph, produced with Graphviz

Spectacle

Another system that represents knowledge in a graphical context is *Spectacle*, one of Aidministrator's core products (Aidministrator is a software provider in the market of information and content management). As stated by Fluit et al (2003), Spectacle "*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 (Fig. 8). With Spectacle, users can efficiently perform analysis, search and navigation, although the system's functionality is restricted to lightweight ontologies only.



Category		
All Pages	484	☐
Vacancies	451	☐
Agriculture ...	0	☐
IT	23	☐
Commerce ...	97	☐
Creative Pr...	6	☐
Finance & ...	90	✓
Healthcare ...	5	☐
Catering	16	☐
Management	23	✓
Education ...	7	☐
Governmen...	1	☐
Human Re...	10	☐
Cleaning	29	☐
Recreation ...	3	☐
Secretarial	10	☐
Technology	166	✓
Logistics	21	☐
Other	28	☐
Other Files	32	☐

Human Resource Manager (http://www.adecco.nl/vacature.asp?id=4695&Page=1)

Fig. 8. Screenshot of a Spectacle cluster map

CPL

CPL (Conceptual Programming Language), introduced by Hartley & Pfeifer (2000), 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 multilabeled bipartite oriented graphs and can express declarative and/or procedural knowledge, by defining object and action constructs (Fig. 9). 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.

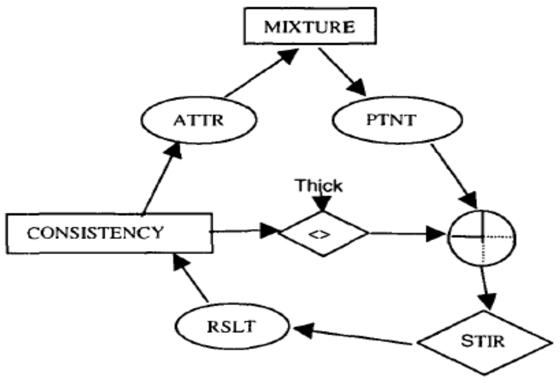


Fig. 9. A simple program, written in CPL

UTILIZING DIGRAPHS FOR RULE REPRESENTATION

Although defeasible reasoning features a significant degree of expressiveness and intuitiveness, it is based on a solid mathematical formulation, which, in many cases, may seem too complicated. So, end users might often consider the conclusion of a defeasible logic theory incomprehensible and complex and, thus, a graphical trace and an explanation mechanism would be very beneficial.

Directed graphs, or *digraphs*, as they are usually referred to, are a special case of graphs that constitute a powerful and convenient way of representing relationships between entities. Usually in a digraph, entities are represented as nodes and relationships as directed lines or arrows that connect the nodes. Each arrow connects only two entities at a time and there can be no two (or more) arrows that connect the same pair of nodes. The orientation of the arrows follows the flow of information in the digraph. Diestel (2000) offers a thorough mathematical definition of directed graphs as well as details on graph theory in general.

Digraphs offer a number of advantages to information visualization, with the most important of them being:

- *comprehensibility*: the information that a digraph contains can be easily and accurately understood by humans and
- *expressiveness*: although the appearance and structure of a digraph may seem simplistic, its topology bears non-trivial information

There are, however, a couple of major disadvantages, not only of digraphs but of graphs in general, as described in (Clarke, 1982):

- it is difficult to associate data of a variety of types with the nodes and with the connections between the nodes in the graph
- as the size and complexity of the graph grows, it becomes more and more difficult to search and modify the graph

On the other hand, in the case of graphical representation of logic rules, digraphs seem to be extremely appropriate, since they offer a number of extra advantages:

- explanation of derived conclusions: the series of inference steps in the graph can be easily detected and retraced
- proof visualization and validation: by going backwards from the conclusion to the triggering conditions, one can validate the truth of the inference result
- especially in the case of defeasible logic rules, the notion of direction can assist in graphical representations of rule attacks, superiorities etc.

Therefore, in this chapter we attempt to exploit the expressiveness and comprehensibility of directed graphs, as well as their suitability for rule representation, but also try to leverage the two aforementioned disadvantages, by adopting a new, "enhanced" digraph approach. This visualization scheme was implemented as part of the VDR-DEVICE system, which is a visual integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies (Bassiliades et al, 2005) for the Semantic Web and is also briefly presented in this work.

Deductive Rules and Digraphs

In an attempt to leverage the first and most important disadvantage of graphs (inability to use a variety of distinct entity types), the digraphs in our approach will contain two kinds of nodes, similarly to the methodology followed by Jantzen (1989). The two node types will be:

- literals, represented by rectangles, which we call "*literal boxes*"
- rules, represented by circles

Thus, according to this principle, the following rule base:

```
p: if A then B
q: if B then ¬C
```

can be represented by the directed graph in Fig. 10.

Each literal box consists of two adjacent "*atomic formula boxes*", with the upper one representing a positive atomic formula and the lower one representing a negated atomic formula. This way, the atomic formulas are depicted together clearly and separately, maintaining their independence.

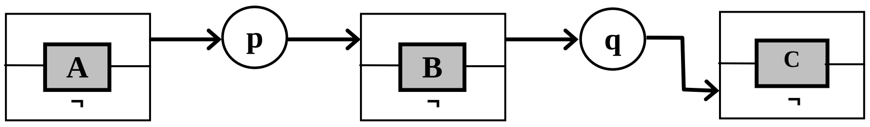


Fig. 10. Digraph displaying a simple rule base

In the case of a rule body that consists of a conjunction of literals the representation is not profoundly affected, as illustrated in Fig. 11.

```
r: if ¬A and B then C
```

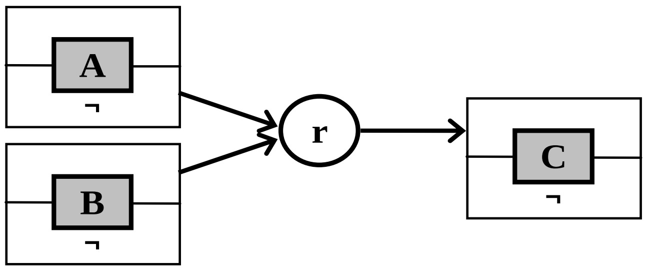


Fig. 11. Digraph featuring a conjunction

As can be observed, digraphs, "*enhanced*" with the addition of distinct node types, offer a significant level of expressiveness in representing rules. The next step is to use directed graphs in the representation of defeasible logic rules, which are more demanding in representational capabilities.

Defeasible Logics And Digraphs

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, >).

In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters. In our approach, each one of the three rule types will be mapped to one of three distinct connection types (i.e. arrows), so that rules of different types can be represented clearly and distinctively.

The first rule type in defeasible reasoning is *strict rules*, which are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, then so is the conclusion. An example of a strict rule is: "*Penguins are birds*", which would become: r_1: `penguin(X)` $\rightarrow$ `bird(X)`. Fig. 12 displays the corresponding digraph.

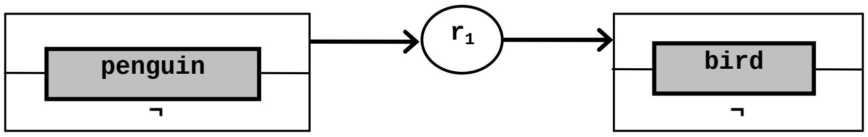


Fig. 12. Digraph displaying a strict rule

Notice that in the rule graph we only represent the predicate and not the literal (i.e. predicate plus all the arguments) because we are mainly interested in making clear to the reader the interrelationships between the concepts (through the rules) and not the complete details of the defeasible theory. The full representation schema is thoroughly described in a later section, with the definition of *class boxes*, *class patterns* and *slot patterns*.

Contrary to strict rules, *defeasible rules* can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. Examples of defeasible rules are r_2: `bird(X)` $\Rightarrow$ `flies(X)`, which reads as: "*Birds typically fly*" and r_3: `penguin(X)` $\Rightarrow$ `¬flies(X)`, namely: "*Penguins typically do not fly*". Rules r_2 and r_3 would be mapped to the directed graph in Fig. 13.

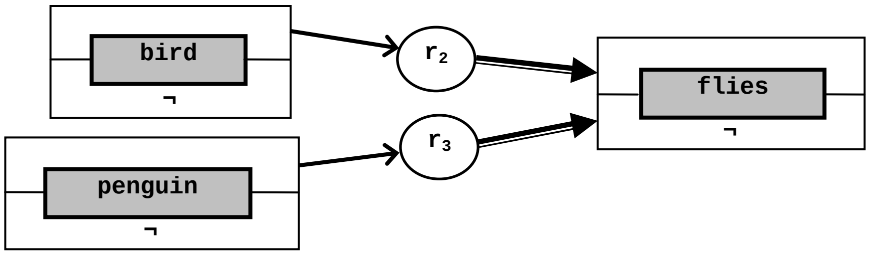


Fig. 13. Digraph displaying two defeasible rules

Defeaters, denoted by $A \sim> p$, are rules that do not actively support conclusions, but can only prevent some of them. In other words, they are used to defeat some defeasible conclusions by producing evidence to the contrary. An example of such a defeater is: r_4: `heavy(X) ~> ¬flies(X)`, which reads as: "*Heavy things cannot fly*". This defeater can defeat the (defeasible) rule r_2 mentioned above and it can be represented by the digraph in Fig. 14.

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 penguin can fly 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 penguin cannot fly. In this case rule r_3 is called *superior* to r_2 and r_2 *inferior* to r_3. In the case of superiority relationships a fourth connection type is introduced. Thus, the aforementioned superiority relationship would be represented by Fig. 15.

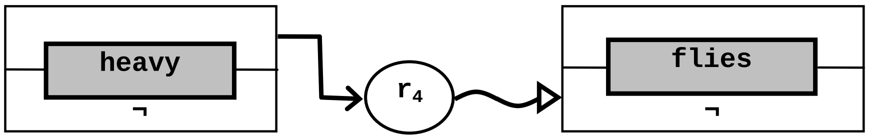


Fig. 14. Digraph displaying a defeater

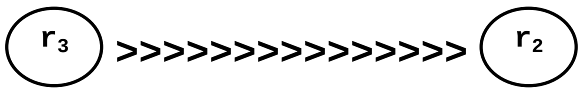


Fig. 15. Digraph displaying a superiority relationship

The set of rules mentioned in this section, namely rules r_1, r_2, r_3 and r_4, form a bigger directed rule graph, which is depicted in Fig. 16.

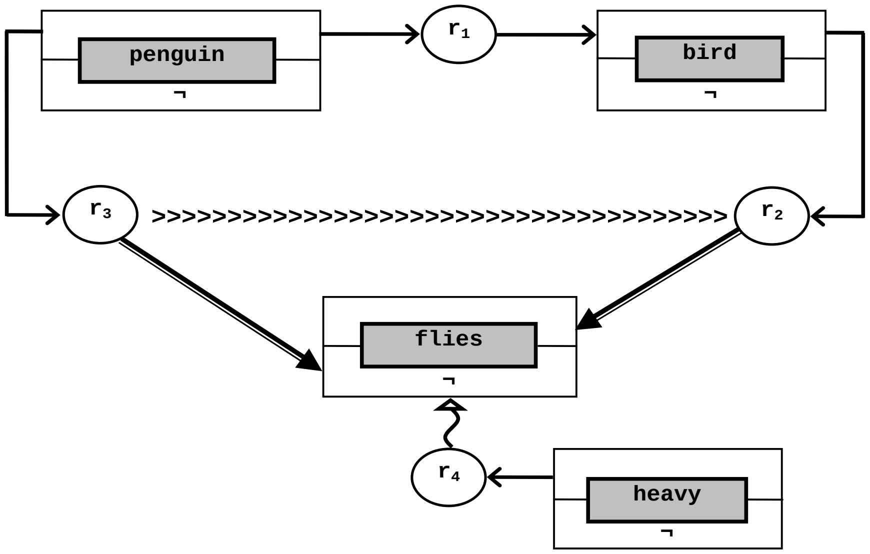


Fig. 16. The digraph formed by the rules r_1, r_2, r_3 and r_4.

Finally, another important type of conflicting evidence in defeasible reasoning is the notion of *conflicting literals*. In some applications, e.g. making an offer in a price negotiation setting, literals are often considered to be conflicting and at most one of a certain set should be derived. Consider the following three rules, which all produce the same literal type as a conclusion, and the constraint that requires at most one of the literals to be true:

```
r₁: a(X) ⇒ p(X)
r₂: b(X) ⇒ p(X)
r₃: c(X) ⇒ p(X)
p(X),p(Y),X≠Y -> ⊥
```

The graph drawn by these rules is depicted in Fig. 17 and, as can be observed, all three rules produce the same result type, which is included in a *single literal truth box*. Of course, superiority relationships could still determine the priorities among the rules.

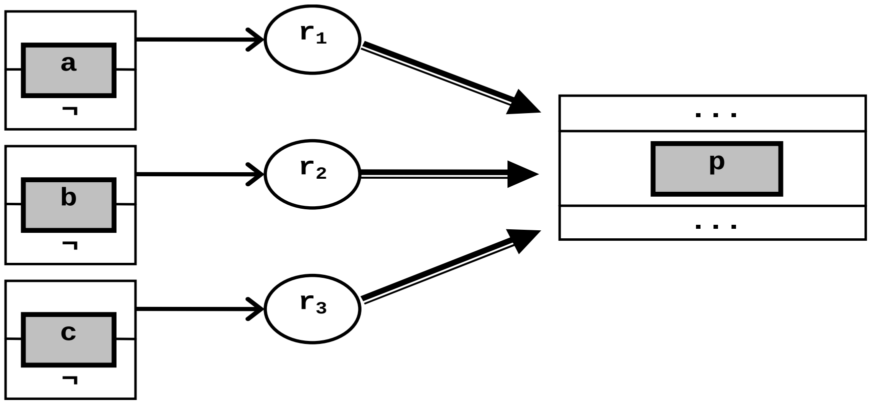


Fig. 17. Representation of conflicting literals as a digraph.

THE VDR-DEVICE SYSTEM

VDR-DEVICE is a visual, integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies (Bassiliades et al, 2005). It consists of two primary components:

1. *DR-DEVICE*, the reasoning system that performs the RDF processing and inference and produces the results, and
2. *DRREd* (Defeasible Reasoning Rule Editor), the rule editor, which serves both as a rule authoring tool and as a graphical shell for the core reasoning system.

Although the two subsystems utilize different technologies and were developed independently, they intercommunicate efficiently, forming a flexible and powerful integrated environment.

The Reasoning System – Architecture And Functionality

The core reasoning system of VDR-DEVICE is DR-DEVICE (Bassiliades et al, 2004) and consists of two primary components (Fig. 19): The *RDF loader/translator* and the *rule loader/translator*. The user can either develop a rule base (program, written in the RuleML-like syntax of VDR-DEVICE – see Fig. 18 for a fragment) with the help of the rule editor described in the following sections, or he/she can load an already existing one, probably developed manually. The rule base contains: (a) a set of rules, (b) the URL(s) of the RDF input document(s), which is forwarded to the RDF loader, (c) the names of the derived classes to be exported as results and (d) the name of the RDF output document.

The rule base is then submitted to the *rule loader* which transforms it into the native CLIPS-like syntax through an XSLT stylesheet and the resulting program is then forwarded to the *rule translator*, where the defeasible logic rules are compiled into a set of CLIPS production rules. This is a two-step process: First, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic deductive rule language, using the translation scheme described in (Bassiliades et al, 2004). Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in (Bassiliades & Vlahavas, 2004). 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).

```
<imp>
      <_rlab ruleID="r1" ruletype="strictrule">
            <ind>r1</ind>
      </_rlab>
      <_head>
            <atom>
                  <_opr>
                        <rel>bird</rel>
                  </_opr>
                  <_slot name="name">
                        <var>X</var>
                  </_slot>
            </atom>
      </_head>
      <_body>
            <atom>
                  <_opr>
                        <rel>penguin</rel>
                  </_opr>
                  <_slot name="name">
                        <var>X</var>
                  </_slot>
            </atom>
      </_body>
</imp>
```

Fig. 18. A strict rule, written in the RuleML-compatible language of DR-DEVICE (this fragment displays rule r_1 of section "**Defeasible Logics And Digraphs**")

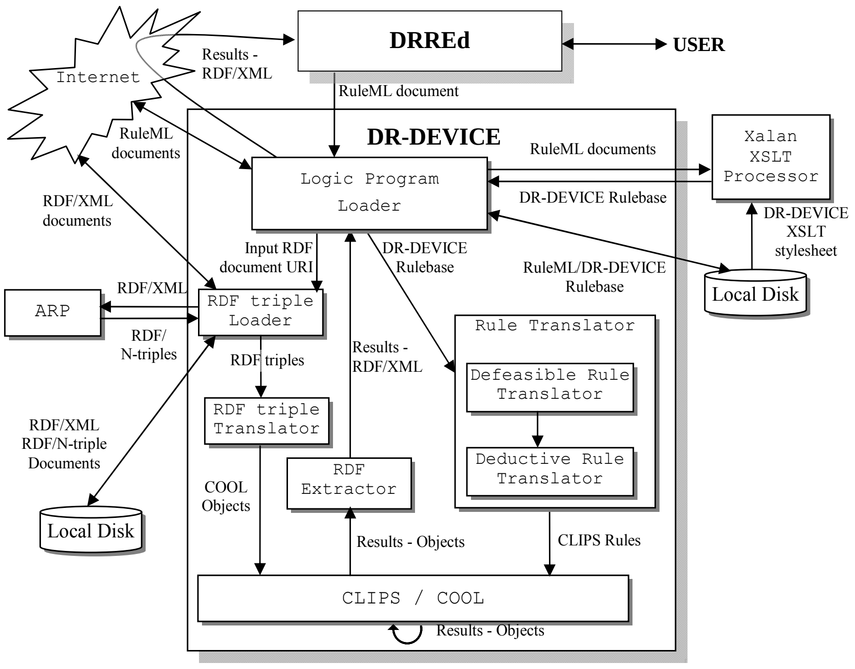
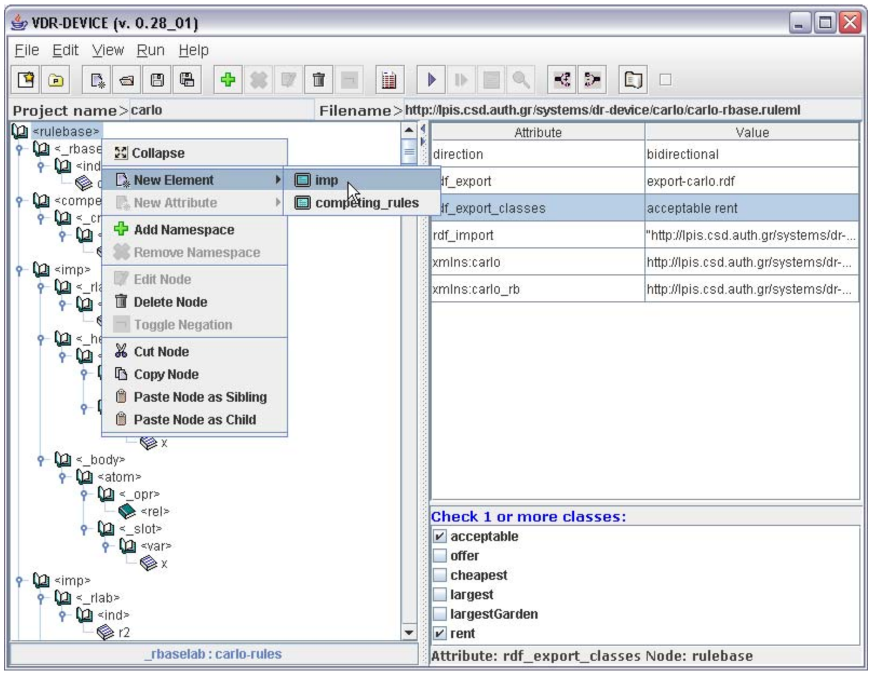


Fig. 19. The VDR-DEVICE system architecture

Meanwhile, the *RDF loader* downloads the input RDF documents, including their schemas, and translates RDF descriptions into CLIPS objects, according to the RDF-to-object translation scheme in (Bassiliades & Vlahavas, 2004), which is briefly described below.
The inference engine of CLIPS performs the reasoning by running the production rules and generates the objects that constitute the result of the initial rule program. The compilation phase guarantees correctness of the reasoning process according to the operational semantics of defeasible logic. Finally, the result-objects are exported to the user as an RDF/XML document through the RDF extractor. The RDF document includes the instances of the exported derived classes, which have been proved.

Rule Editor – Design And Functionality

VDR-DEVICE is equipped with DRREd, a visual rule editor that aims at enhancing user-friendliness and efficiency during the development of VDR-DEVICE RuleML documents. Key features of the software include: (a) functional flexibility - program utilities can be triggered via a variety of overhead menu actions, keyboard shortcuts or popup menus, (b) improved development speed - rule bases can be developed in just a few steps and (c) powerful safety mechanisms – the correct syntax is ensured and the user is protected from syntactic or RDF Schema related semantic errors.



Project name> carlo
Filename> http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-rbase.ruleml

Attribute	Value
direction	bidirectional
df_export	export-carlo.rdf
df_export_classes	acceptable rent
rdf_import	"http://lpis.csd.auth.gr/systems/dr-...
xmlns:carlo	http://lpis.csd.auth.gr/systems/dr-...
xmlns:carlo_rb	http://lpis.csd.auth.gr/systems/dr-...

Check 1 or more classes:
- ✓ acceptable
- ☐ offer
- ☐ cheapest
- ☐ largest
- ☐ largestGarden
- ✓ rent

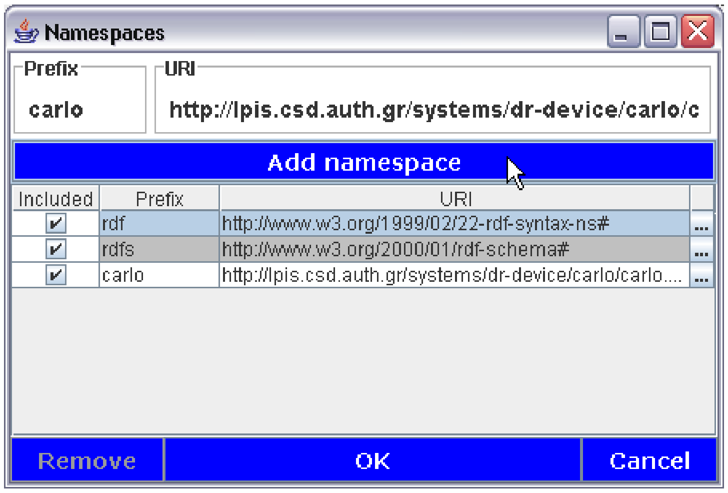
_rbaselab : carlo-rules
Attribute: rdf_export_classes Node: rulebase

Fig. 20. The graphical rule editor main window

The main window of the program (Fig. 20) is composed of two major parts: a) the upper part includes the menu bar, which contains the program menus, and the toolbar that includes icons, representing the most common utilities of the rule editor, and b) the central and more “bulky” part is the primary frame of the main window and is in turn divided in two panels:
The left panel displays the rule base in XML tree-like format. The user can navigate through the tree and can add to or remove elements from the tree, obeying DTD constraints. Furthermore, the operations allowed on each element depend on the element's meaning within the rule tree.

The right panel shows a table, which contains the attributes that correspond to the selected tree node in the left panel. The user can also perform editing functions on the attributes, by altering the value for each attribute in the panel that appears in the bottom-right of the main window, below the attributes table. The values that the user can insert depend on the chosen attribute. The development of a rule base using VDR-DEVICE is a process that depends heavily on the context, i.e. the node being edited. Some examples follow:

- When a new element is added to the tree, all its mandatory sub-elements are also added. In cases, where 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. 20).
- The `atom` element is treated in a specific manner, since it 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.
- The function names in a `fun_call` element are partially constrained by the list of CLIPS built-in function. However, the user can still use a user-defined function, which is unconstrained.
- 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 (e.g. superior).



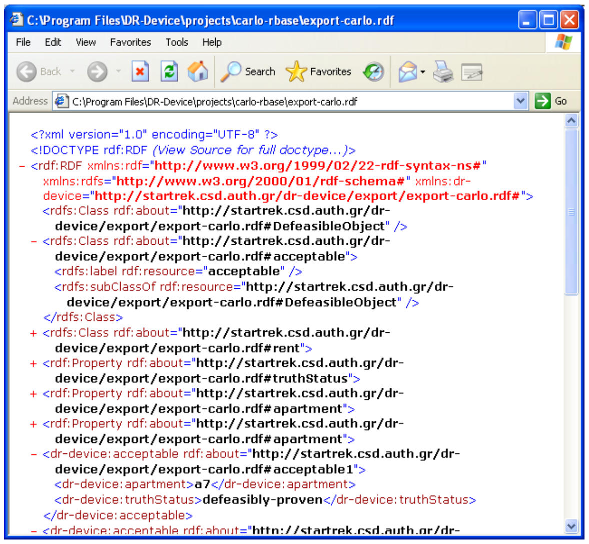
Prefix: carlo
URI: http://lpis.csd.auth.gr/systems/dr-device/carlo/c

Included	Prefix	URI
✓	rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
✓	rdfs	http://www.w3.org/2000/01/rdf-schema#
✓	carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo....

Fig. 21. The namespace dialog window (NDW)

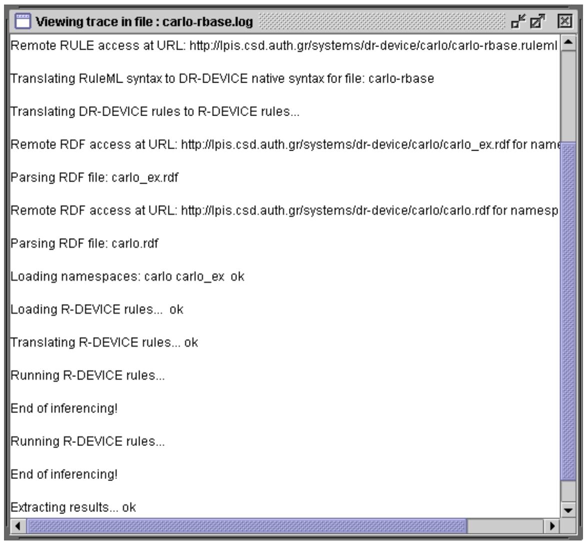
An important component of the editor is the *namespace dialog window* (NDW, see Fig. 21), where the user can determine which RDF/XML namespaces will be used by the rule base. Actually, namespaces are treated as addresses of input RDF Schema ontologies that contain the vocabulary for the input RDF documents, over which the rules will be run. The namespaces that have been manually selected by the user to be included in the system are analyzed in order to extract all the allowed class and property names for the rule base being developed. 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. Furthermore, 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. 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 an Internet Explorer window, launched by the system (Fig. 22). They can also examine the execution trace of compilation and running, both at run-time and also after the whole process has been terminated (Fig. 23).


C:\Program Files\DR-Device\projects\carlo-rbase\export-carlo.rdf
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Address
C:\Program Files\DR-Device\projects\carlo-rbase\export-carlo.rdf
Go

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
- <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:dr-
    device="http://startrek.csd.auth.gr/dr-device/export/export-carlo.rdf#">
    <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#DefeasibleObject" />
  - <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#acceptable">
      <rdfs:label rdf:resource="acceptable" />
      <rdfs:subClassOf rdf:resource="http://startrek.csd.auth.gr/dr-
        device/export/export-carlo.rdf#DefeasibleObject" />
    </rdfs:Class>
  + <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#rent">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#truthStatus">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#apartment">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#apartment">
  - <dr-device:acceptable rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#acceptable1">
      <dr-device:apartment>a7</dr-device:apartment>
      <dr-device:truthStatus>defeasibly-proven</dr-device:truthStatus>
    </dr-device:acceptable>
  - <dr-device:acceptable rdf:about="http://startrek.csd.auth.gr/dr-
```

Fig. 22. The Trace window



```
Remote RULE access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-rbase.ruleml

Translating RuleML syntax to DR-DEVICE native syntax for file: carlo-rbase

Translating DR-DEVICE rules to R-DEVICE rules...

Remote RDF access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo_ex.rdf for name

Parsing RDF file: carlo_ex.rdf

Remote RDF access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf for namesp

Parsing RDF file: carlo.rdf

Loading namespaces: carlo carlo_ex  ok

Loading R-DEVICE rules...  ok

Translating R-DEVICE rules... ok

Running R-DEVICE rules...

End of inferencing!

Running R-DEVICE rules...

End of inferencing!

Extracting results... ok
```

Fig. 23. The Results window

DRREd's Digraph Module

DRREd is equipped with a module that allows the creation of a directed rule graph from the defeasible rule base developed by the editor. This way, users are offered an extra means of visualizing the rule base, besides XML-tree format and, thus, possess a better aspect of the rule base displayed and the inference results produced. This section describes thoroughly the steps towards the generation of the derived graph.

Collecting the Base and Derived Classes

The RDF Schema documents contained in the namespace dialog window are being parsed, using the ARP parser of Jena (McBride, 2001). The names of the classes found are collected in the *base class vector* (CV_b), which already contains `rdfs:Resource`, the superclass of all RDF user classes. Therefore, the CV_b vector is constructed as follows:

`rdfs:Resource` $\in CV_b \wedge (\forall C$ (C `rdf:type rdfs:Class`) $\rightarrow C \in CV_b)$

where (X Y Z) represents an RDF triple found in the RDF Schema documents.

Except from the base class vector, there also exists the *derived class vector* (CV_d), which contains the names of the derived classes, i.e. the classes which lie at rule heads (*conclusions*). CV_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. This vector 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.

The union of the above two vectors results in CV_f, which is the *full class vector* ($CV_f = CV_b \cup CV_d$) and it is used for constraining the allowed class names, when editing the contents of the `rel` element inside atom elements of the rule body.

Elements of the Rule Graph

For each class that belongs to the full class vector CV_f, a *class box* is constructed, which, apparently, is simply a container. The class boxes are initially empty and are dynamically populated with one or more *class patterns*. For each `atom` element inside a rule head or body, a new class pattern is created and is 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. In practice, class patterns express conditions on instances of the specific class. Nevertheless, a class box could indeed remain empty, in the case when 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/removed from the full class vector, as mentioned in the previous section. Empty class boxes still appear in the rule graph, but naturally play a limited role.

Visually, class patterns appear as literal boxes, whose design was thoroughly described in a previous section of this chapter (see "*Deductive Rules and Digraphs*"). Similarly to class boxes, class patterns are empty, when they are initially created and are soon populated with one or more *slot patterns*. For each `_slot` element inside an atom, a slot pattern is created that consists of a slot name (contained inside the corresponding attribute) and, optionally, a variable and a list of value constraints. The variable is used in order for the slot value to be unified, with the latter having to satisfy the list of constraints. In other words, slot patterns represent conditions on slots

(or *class properties*). Each of the slot pattern "ingredients" is being retrieved from the - direct and indirect - children of the `_slot` element in the XML tree representation of the rule base. Since RuleML atoms are actually atomic formulas (i.e. they correspond to queries over RDF resources of a certain class with certain property values), the truth value associated with each returned class instance will be either positive or negative. This justifies the mapping of class patterns to literal boxes.
An example of all the above can be seen in Fig. 24 and Fig. 25. The figures illustrate a class box that contains three class patterns applied on the *person* class and a code fragment matching the third class pattern, written in the RuleML-like syntax of VDR-DEVICE.

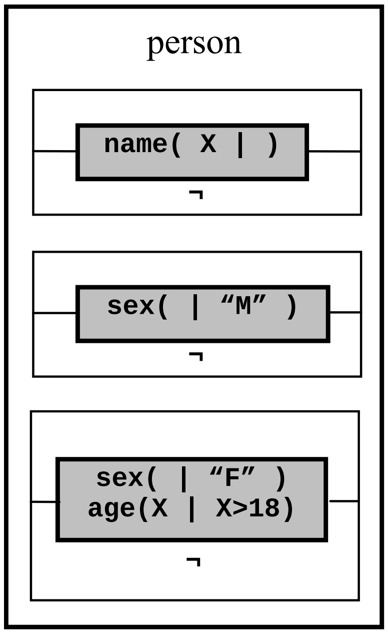


Fig. 24. A class box example

```
<atom>
  <_opr>
    <rel href="person"/>
  </_opr>
  <_slot name="sex">
    <ind>"F"</ind>
  </_slot>
  <_slot name="age">
    <_and>
      <var>x</var>
      <function_call name="&gt;">
        <var>x</var>
        <ind>18</ind>
      </function_call>
    </_and>
  </_slot>
</atom>
```

Fig. 25. A code fragment for the third class pattern of Fig. 24

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 into 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 right-hand side is utilized; thus, the second class pattern of the box in Fig. 24, for example, refers to all the *male* persons. This way the content of the slot arguments is clearly depicted and easily comprehended.
Besides class boxes and their "ingredients" (class patterns, slot patterns), a number of additional graph elements exists: circles that represent rules and arrows that connect the nodes in the graph. The graphical representation of rules in the digraph, using circles, was described in section "*Deductive Rules and Digraphs*". The circles in the graph include the rule name, which is equal to the value of the `ruleID` attribute in the `_rlab` element of the corresponding rule. As for the arrows in the graph, there exist four types of them: three for the rule type and one for the superiority relationship, as explained in section "*Defeasible Logics And Digraphs*". 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`). As for the superiority relationship, it is represented as attribute (`superior` attribute) inside the superior rule element.

Building the Rule Graph

After having collected all the necessary graph elements and having populated all the class boxes with the appropriate class and slot patterns, three new vectors are created:

- The *base class boxes* vector CB_b that contains the class boxes corresponding to base classes.
- The *derived class boxes* vector CB_d that contains the class boxes corresponding to derived classes.
- The vector *R* that includes all the rules of the rule base.

An important matter arises at this point, regarding the placement of each element in the graph; namely, a plan has to be drawn, concerning the location of the elements in the graph drawing panel of VDR-DEVICE. For this affair, a variation of the rule stratification algorithm found in (Ullman, 1988) was implemented. The algorithm aims at giving a left-to-right orientation to the flow of information in the graph; i.e. the arcs in the digraph are directed from left to right, making the derived graph less complex, more comprehensible and easily readable. The rule stratification algorithm implemented is displayed in Fig. 26.

```
1.Assign all class boxes contained in CBb to stratum 1
2.Assign all rules in R, whose bodies contain only
class patterns belonging to a base class box, to
stratum 2
3.Remove rules applied in STEP 2 from R
4.i = 3
WHILE R.size > 0 {
      5.Assign the class boxes contained in CBd that
      contain class patterns that act as heads for
      rules in stratum i - 1 to stratum i
      6.All rules in R with bodies in stratum i are
      assigned to stratum i + 1
      7.Remove rules applied in previous STEP from R
      8.i = i + 1 }
```

Fig. 26. The rule stratification algorithm

An example that could better illustrate the functionality of the algorithm is adopted from (Antoniou & van Harmelen, 2004, pp. 161-165). The example deals with apartment renting; a number of available apartments reside in an RDF document along with the properties of each apartment and the potential renter expresses his/her requirements in defeasible logic, regarding the apartment he/she wishes to rent. A defeasible rule base that contains the requirements of an imaginary renter is included in the example. The classes and the rules included in the rule base are displayed in Table 1.

Table 1. Base classes, derived classes and rules included in the rule base of the example

Base Classes	`carlo:apartment`

Derived Classes	`acceptable`
	`offer`
	`cheapest`
	`largest`
	`largestGarden`

Rules	r_1 - r_{12}
	`find_cheapest`
	`find_largest`
	`find_largestGarden`

Six strata are needed to display all the rules. Table 2 shows the stratum assignments, according to the algorithm.

Table 2. Stratum assignments for the classes and the rules in the example

stratum 1	`carlo:apartment`
stratum 2	r_1 to r_8
stratum 3	`acceptable` `offer`

stratum 4	r_9
	`find_cheapest`
	`find_largest`
	`find_largestGarden`

stratum 5	`cheapest`
	`largest`
	`largestGarden`

stratum 6	r_{10} r_{11} r_{12}

In our approach each stratum is considered as a *column* in the graph drawing panel of the system. Thus, the first stratum is mapped to the first column on the left, the second stratum to the column on the right of the first one and so on. Nodes in one column are never connected with nodes in the same column. Note also that no two nodes of the same type are ever connected with an arrow. As a result, the digraphs produced by our approach can be considered a type of *bipartite graphs*.
Obviously, the number of strata/columns can be too big to fit the screen. In this case the user can traverse the graph, using the scroll bars located on the edges of the panel. Furthermore, similarly to the XML-tree format of the rule base, in the digraph there is also the possibility to collapse or

expand certain parts of it. This way, a twofold advantage is offered: (a) the complexity of the digraph is minimized, since only a limited number of graph parts are visible at a time and (b) the level of comprehensibility on behalf of the user is raised, since he/she does not have to focus on the whole graph, but only a part of it.
The two aspects of the rule base, namely the XML-tree and the directed graph are interrelated, meaning that traversal and alterations in one will also be depicted in the other. 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 will also be selected and the data relevant to it displayed.
As mentioned in a previous section, the VDR-DEVICE system has a CLIPS-like syntax and a RuleML-compliant one. Fig. 27 illustrates the first four rules of the above example expressed in the former syntax (the latter syntax can be found in APPENDIX A). Fig. 28 displays the same rules in the typical defeasible syntax.

```
(defeasiblerule r1
       (carlo:apartment (carlo:name ?x))
  =>
       (acceptable (apartment ?x))
)

(defeasiblerule r2
       (declare (superior r1))
       (carlo:apartment    (carlo:name ?x)
                           (carlo:bedrooms  ?y & :(<  ?y 2)))
  =>
       (not (acceptable (apartment ?x)))
)

(defeasiblerule r3
       (declare (superior r1))
       (carlo:apartment    (carlo:name ?x)
                           (carlo:size  ?y & :(<  ?y 45)))
  =>
       (not (acceptable (apartment ?x)))
)

(defeasiblerule r4
       (declare (superior r1))
       (carlo:apartment    (carlo:name ?x)
                           (carlo:pets "no"))
  =>
       (not (acceptable (apartment ?x)))
)
```

Fig. 27. Rules r_1-r_4 expressed in the CLIPS-like syntax of VDR-DEVICE

```
r1 : ⇒ acceptable(X)
r2 : bedrooms(X, Y), Y < 2 ⇒ ¬acceptable(X)
r3 : size(X, Y), Y < 45 ⇒ ¬acceptable(X)
r4 : ¬pets(X) ⇒ ¬acceptable(X)
```

Fig. 28. Rules r_1-r_4 expressed in defeasible syntax

Fig. 29 displays the output of the VDR-DEVICE rule graph drawing module, when a portion of the rule base of the example in (Antoniou & van Harmelen, 2004, pp. 161-165) is loaded. For reasons of simplicity, only the first four rules are applied (rules r_1-r_4) and there exist only one base and one derived class. The total number of strata in *this* case is three (i.e. three columns).

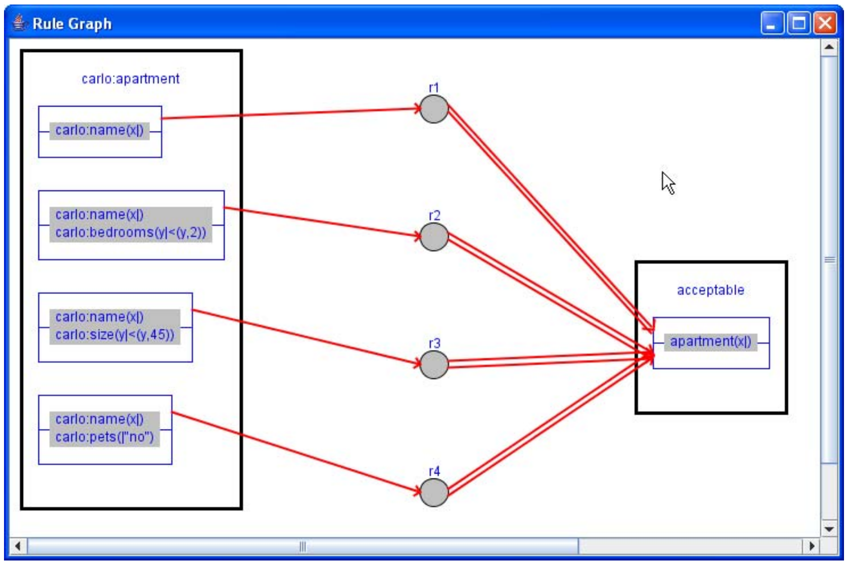


Fig. 29. The rule graph drawing module of DRREd

CONCLUSIONS AND FUTURE WORK

This chapter presented the utilization of *directed graphs* (or *digraphs*) in the representation of defeasible logic rules, an affair where digraphs proved to be extremely suitable. It is, however, difficult to associate data of a variety of types with the nodes and with the connections between the nodes in the graph and this is probably the major disadvantage of graphs in information visualization.

In our approach we attempted to apply the use of digraphs in the representation of defeasible logic rules, exploiting the advantages they offer (comprehensibility, expressiveness), but also trying to leverage their primary disadvantage. We adopted a novel "*enhanced*" digraph approach that features two types of nodes (one for rules and one for literals) and four connection types (one for each of the three distinct rule types in defeasible reasoning and a fourth for the representation of superiority relationships). It was observed that these enhanced directed graphs offer a significant level of expressiveness in representing defeasible logic rules.

A system that implements this digraph approach was briefly described. The system is called *VDR-DEVICE* and is a visual, integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies. VDR-DEVICE consists of two subcomponents: *DR-DEVICE*, the reasoning system that performs the inference and produces the results and *DRREd*, the rule editor, which serves both as a rule authoring tool and as a graphical shell for DR-DEVICE. DRREd's graph drawing module that allows the creation of a directed rule graph from the defeasible rule base developed by the editor was also demonstrated. The steps towards the creation of a digraph from a rule base include:

1. *collecting the classes*, meaning that all the base classes and all the derived class are detected and collected in a vector,
2. *building the nodes and connection types of the rule graph*, according to the information collected,
3. *generating the rule graph*, using a stratification algorithm to determine the location of the graph elements.

In the future, we plan to delve deeper into the proof layer of the Semantic Web architecture by enhancing further the rule representation utility demonstrated with rule execution tracing, explanation, proof exchange in an XML or RDF format, proof visualization and validation, etc. These facilities would be useful for increasing the trust of users for the Semantic Web agents and for automating proof exchange and trust among agents in the Semantic Web.

REFERENCES

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

Antoniou, G. (2002). Nonmonotonic Rule Systems on Top of Ontology Layers. *Proc. 1st Int. Semantic Web Conference* (pp. 394-398). LNCS 2342. Springer-Verlag.

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

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

Antoniou, G., & Harmelen, F. van. (2004). *A Semantic Web Primer*. MIT Press.

Antoniou, G., Skylogiannis, T., Bikakis, A., & Bassiliades, N. (2005). DR-BROKERING – A Defeasible Logic-Based System for Semantic Brokering. *Proc. IEEE Int. Conference on E-Technology, E-Commerce and E-Service*. 414-417. IEEE Computer Society.

Ashri, R., Payne, T., Marvin, D., Surridge, M., & Taylor, S. (2004). Towards a Semantic Web Security Infrastructure. *Proc. Semantic Web Services 2004 Spring Symposium Series*. Stanford University, Stanford California.

Bassiliades, N., Antoniou, G., & Vlahavas, I. (2004). A Defeasible Logic Reasoner for the Semantic Web, *Proc. RuleML 2004* (pp. 49-64). Springer-Verlag, LNCS 3323. Hiroshima, Japan.

Bassiliades, N., Kontopoulos, E. & Antoniou, G. (2005). A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web. *Proc. International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)* (pp. 172-186). A. Adi, S. Stoutenburg, S. Tabet (Ed.). Springer-Verlag. LNCS 3791. Galway, Ireland.

Bassiliades, N., & Vlahavas, I. (2004). R-DEVICE: A Deductive RDF Rule Language. *Proc. RuleML 2004* (pp. 65-80). Springer-Verlag. LNCS 3323. Hiroshima, Japan.

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

Clarke, D. (1982). An Augmented Directed Graph Base for Application Development. *Proc. 20th annual Southeast regional conference* (pp. 155-159). ACM Press. Knoxville, Tennessee, USA.

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

Fluit, C., Sabou, M. & Harmelen, F. van. (2003). Ontology-Based Information Visualization. In V. Geroimenko and C. Chen (Ed.) *Visualizing the Semantic Web* (pp. 36-48). Springer-Verlag.

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

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

Graphviz - Graph Visualization Software, Retrieved April 23, 2006, from http://www.graphviz.org.

Grosof, B.N., Gandhe, M.D., & Finin, T.W. (2002). 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.

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

Hartley, R. & Pfeiffer, H. (2000). Visual Representation of Procedural Knowledge. *Proceedings of the 2000 IEEE international Symposium on Visual Languages (Vl'00)*. VL. IEEE Computer Society, Washington, DC, 63.

Jantzen, J. (1989). Inference Planning Using Digraphs and Boolean Arrays. *Proc. International Conference on APL* (pp. 200-204). ACM Press.

Kehler, T.P., & Clemenson G.D. KEE the knowledge engineering environment for industry. *Systems and Software*, 3(1):212-- 224, January 1984.

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

McBride, B. (2001). Jena: Implementing the RDF Model and Syntax Specification. *Proc. 2nd Int. Workshop on the Semantic Web.*

Nute, D. (1987). Defeasible Reasoning. *Proc. 20th Int. Conference on Systems Science* (pp. 470-477). IEEE Press.

Powers, S. (2003). *Practical RDF*. Beijing. Cambridge: O'Reilly.

Shalfield, R. (2005). *VisiRule User Guide*. Retrieved March 10, 2006, from http://www.lpa.co.uk/ftp/4600/vsr_ref.pdf

Steel, B. D. (2005). *WIN-PROLOG Technical Reference*. Retrieved April 23, 2006, from http://www.lpa.co.uk/ftp/4600/win_ref.pdf

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

APPENDIX A

Rules r_1 to r_4 expressed in the RuleML-compliant syntax of VDR-DEVICE:

```
<imp>
    <_rlab ruleID="r1" ruletype="defeasiblerule">
      <ind href="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-
rbase.ruleml#r1">r1</ind>
    </_rlab>
    <_head>
      <atom>
        <_opr>
          <rel href="acceptable"/>
        </_opr>
        <_slot name="apartment">
          <var>x</var>
        </_slot>
      </atom>
    </_head>
    <_body>
      <atom>
        <_opr>
          <rel href="carlo:apartment"/>
        </_opr>
        <_slot name="carlo:name">
          <var>x</var>
        </_slot>
      </atom>
    </_body>
  </imp>
  <imp>
    <_rlab ruleID="r2" ruletype="defeasiblerule" superior="r1">
      <ind href="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-
rbase.ruleml#r2">r2</ind>
    </_rlab>
    <_head>
      <neg>
        <atom>
          <_opr>
            <rel href="acceptable"/>
          </_opr>
          <_slot name="apartment">
            <var>x</var>
          </_slot>
        </atom>
      </neg>
    </_head>
    <_body>
      <atom>
        <_opr>
          <rel href="carlo:apartment"/>
        </_opr>
        <_slot name="carlo:name">
          <var>x</var>
        </_slot>
        <_slot name="carlo:bedrooms">
          <_and>
            <var>y</var>
            <function_call name="&lt;">
              <var>y</var>
              <ind>2</ind>
            </function_call>
          </_and>
        </_slot>
      </atom>
    </_body>
  </imp>
  <imp>
    <_rlab ruleID="r3" ruletype="defeasiblerule" superior="r1">
      <ind href="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-
rbase.ruleml#r3">r3</ind>
    </_rlab>
    <_head>
      <neg>
        <atom>
          <_opr>
            <rel href="acceptable"/>
          </_opr>
          <_slot name="apartment">
            <var>x</var>
          </_slot>
        </atom>
      </neg>
    </_head>
    <_body>
      <atom>
        <_opr>
          <rel href="carlo:apartment"/>
        </_opr>
        <_slot name="carlo:name">
          <var>x</var>
        </_slot>
        <_slot name="carlo:size">
          <_and>
            <var>y</var>
            <function_call name="&lt;">
              <var>y</var>
              <ind>45</ind>
            </function_call>
          </_and>
        </_slot>
      </atom>
    </_body>
  </imp>
  <imp>
    <_rlab ruleID="r4" ruletype="defeasiblerule" superior="r1">
      <ind href="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-
rbase.ruleml#r4">r4</ind>
    </_rlab>
    <_head>
      <neg>
        <atom>
          <_opr>
            <rel href="acceptable"/>
          </_opr>
          <_slot name="apartment">
            <var>x</var>
          </_slot>
        </atom>
      </neg>
    </_head>
    <_body>
      <atom>
        <_opr>
          <rel href="carlo:apartment"/>
      </_opr>
      <_slot name="carlo:name">
        <var>x</var>
      </_slot>
      <_slot name="carlo:pets">
        <ind>"no"</ind>
      </_slot>
    </atom>
  </_body>
</imp>
```