A Defeasible Logic Reasoner for the Semantic Web

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

[1]Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
{nbassili, vlahavas}@csd.auth.gr
[2]Institute of Computer Science, FO.R.T.H.
P.O. Box 1385, GR-71110, Heraklion, Greece
antoniou@ics.forth.gr

Abstract. Defeasible reasoning is a rule-based approach for efficient reasoning with incomplete and inconsistent information. Such reasoning is, among others, useful for ontology integration, where conflicting information arises naturally; and for the modeling of business rules and policies, where rules with exceptions are often used. This paper describes these scenarios in more detail, and reports on the implementation of a system for defeasible reasoning on the Web. The system is called DR-DEVICE and is capable of reasoning about RDF metadata over multiple Web sources using defeasible logic rules. The system is implemented on top of CLIPS production rule system and builds upon R-DEVICE, an earlier deductive rule system over RDF metadata that also supports derived attribute and aggregate attribute rules. Rules can be expressed either in a native CLIPS-like language, or in an extension of the OO-RuleML syntax. The operational semantics of defeasible logic are implemented through compilation into the generic rule language of R-DEVICE. The paper also presents a full semantic web broker example for apartment renting.

Keywords: RDF, rules, reasoning, defeasible logic, rule markup languages, semantic brokering

1. Introduction

The development of the Semantic Web [16] proceeds in layers, each layer being on top of other layers. At present, the highest layer that has reached sufficient maturity is the ontology layer in the form of the description logic based languages of DAML+OIL [20] and OWL [45].

The next step in the development of the Semantic Web will be the logic and proof layers, and rule systems appear to lie in the mainstream of such activities. Moreover, rule systems can also be utilized in ontology languages. So, in general rule systems can play a twofold role in the Semantic Web initiative: (a) they can serve as extensions of, or alternatives to, description logic based ontology languages; and (b) they can be used to develop declarative systems on top of (using) ontologies. Reasons why rule systems are expected to play a key role in the further development of the Semantic Web include the following:

- Seen as subsets of predicate logic, monotonic rule systems (Horn logic) and description logics are orthogonal; thus they provide additional expressive power to ontology languages.
- Efficient reasoning support exists to support rule languages.
- Rules are well known in practice, and are reasonably well integrated in mainstream information technology.

Possible interactions between description logics and monotonic rule systems were studied in [27]. Work on hybrid reasoning [32] showed the computational difficulties involved in the combination of description logics and Horn rules. Recent works in this area include [40], [30].

This paper is devoted to a different problem, namely conflicts among rules. Here we just mention the main sources of such conflicts, which are further expanded in section 2. At the ontology layer: (a) default inheritance within ontologies, (b) ontology merging; and at the logic and reasoning layers: (a) rules with exceptions as a natural representation of business rules, (b) reasoning with incomplete information.

Defeasible reasoning is a simple rule-based approach to reasoning with incomplete and inconsistent information. It can represent facts, rules, and priorities among rules. This reasoning family comprises defeasible logics ([41], [6]) and Courteous Logic Programs [26]. The main advantage of this approach is the combination of two desirable features: enhanced representational capabilities allowing one to reason with incomplete and contradictory information, coupled with low computational complexity compared to mainstream nonmonotonic reasoning.

In this paper we report on the implementation of DR-DEVICE which is a defeasible reasoning system for the Semantic Web. The most important features of DR-DEVICE are the following:

- It supports multiple rule types of defeasible logic, such as strict rules, defeasible rules, and defeaters. Furthermore, it supports priorities among rules.
- It supports two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals.
- Its user interface is compatible with RuleML [17], the main standardization effort for rules on the Semantic Web.
- It supports direct import from the Web and processing of RDF data and RDF Schema ontologies.
- It supports direct export to the Web of the results (conclusions) of the logic program as an RDF document.
- It is built on-top of a CLIPS-based implementation of deductive rules ([12], [13]). The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes. However, the implementation is declarative because it interprets the not operator using Well-Founded Semantics [23].

As a result of the above, DR-DEVICE is a powerful declarative system supporting

- rules, facts and ontologies;
- major Semantic Web standards: RDF, RDFS, RuleML;
- monotonic and nonmonotonic rules, reasoning with inconsistencies.

In the rest of this paper we detail on various motivating cases for using conflicting rules on the Semantic Web in section 2; in section 3 we briefly introduce the syntax and semantics of defeasible logics; in section 4 we present the architecture of the DR-DEVICE system, including a brief description of the R-DEVICE sys-

tem which lies at the core. Section 5 describes the syntax of defeasible logic rules in DR-DEVICE and its RuleML syntax; Section 6 details the translation scheme from the defeasible logic rule language of DR-DEVICE into the deductive rule language of R-DEVICE; Section 7 presents the performance evaluation we performed on DR-DEVICE using defeasible theories of various types and sizes; Section 8 presents a full use case of a semantic web broker that reasons about apartment renting, using defeasible logic rules. Finally, section 9 briefly overviews related work and section 10 concludes this paper and poses future research directions.

2. Motivation for Conflicting Rules on the Semantic Web

In this section we describe in more detail certain scenarios that justify the need for defeasible reasoning on the Semantic Web.

Reasoning with Incomplete Information. In [3] 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 which 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 classical case for the use of nonmonotonic knowledge representation and reasoning [38].

Rules with Exceptions. Rules with exceptions are a natural representation for policies and business rules [5]. And priority information is often implicitly or explicitly available to resolve conflicts among rules. Potential applications include security policies ([11], [33]), business rules [2], personalization, brokering, bargaining, and automated agent negotiations [22], and electronic contracts [24].

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 [25] 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, nonmonotonic rule systems can be utilized in ontology languages.

Ontology and Knowledge Merging. When ontologies from different authors and/or sources are merged, contradictions arise naturally. Moreover, in domain 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 [27]) 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 nonmonotonic 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. 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, nonmonotonic rule systems can support ontology integration.

3. Defeasible Logics

3.1. Basic Characteristics

The root of defeasible logics lies on research in knowledge representation, and in particular on inheritance networks. Defeasible logics can be seen as inheritance networks expressed in a logical rules language. In fact, they are the first nonmonotonic reasoning approach designed from its beginning to be implementable.

Being nonmonotonic, defeasible logics deal with potential conflicts (inconsistencies) among knowledge items. Thus they contain classical negation, contrary to usual logic programming systems. They can also deal with negation as failure (NAF), the other type of negation typical of nonmonotonic logic programming systems; in fact, [44] argues that the Semantic Web requires both types of negation. In defeasible logics, often it is assumed that NAF is not included in the object language. However, as [6], [10] show, it can be easily simulated when necessary. Thus, we may use NAF in the object language and transform the original knowledge to logical rules without NAF exhibiting the same behavior.

Conflicts among rules are indicated by a conflict between their conclusions. These conflicts are of local nature. The simpler 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.

Defeasible logics are skeptical 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. This is, among others, achieved through the absence of disjunction and 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.

Generally speaking, defeasible logics are closely related to Courteous Logic Programs [26], [28]; the latter were developed much later than defeasible logics. DLs have the following advantages:

- They have more general semantic capabilities, e.g. in terms of loops, ambiguity propagation etc.
- They have been studied much more deeply, with strong results in terms of proof theory [6], semantics [36] and computational complexity [34]. As a consequence, its translation into logic programs, a cornerstone of DR-DEVICE, has also been studied thoroughly [7], [37].

However, Courteous Logic Programs have also had some advantages:

- They adopted the idea of mutually exclusive literals, an idea incorporated in DR-DEVICE.

- They allow access to procedural attachments, something we have chosen not to study in our work so far, although clearly procedural function calls to the underlying CLIPS system are possible in DR-DEVICE, at the loss of declarativity.

In the following we discuss in more detail some of the ideas and concepts mentioned here.

3.2. The Language

A *defeasible theory D* is a triple (F, *R*,>) where F is a finite set of facts, *R* a finite set of rules, and > a superiority relation on R. Rules containing free variables are interpreted as the set of their variable-free instances. Facts are ground atoms.

There are three kinds of rules: *Strict rules* are denoted by $A \rightarrow p$, and are interpreted in the classical sense: whenever the premises are indisputable then so is the conclusion. An example of a strict rule is "Professors are faculty members", written formally as: `professor(X)` $\rightarrow$ `faculty(X)`. Inference from strict rules only is called *definite inference*. Strict rules are intended to define relationships that are definitional in nature, for example ontological knowledge.

Defeasible rules are denoted by $A \Rightarrow p$, and can be defeated by contrary evidence. An example of such a rule is `faculty(X)` $\Rightarrow$ `tenured(X)` which reads as follows: "Faculty members are typically tenured".

Defeaters are denoted as $A \sim> p$ and are used only to prevent some conclusions, not to actively support conclusions. An example of such a defeater is `assistantProf(X)` ~> ¬`tenured(X)` which reads as follows: "Assistant professors may be not tenured".

A *superiority relation* on R is an acyclic relation > on R (that is, the transitive closure of > is irreflexive). When $r_1 > r_2$, then r_1 is called *superior* to r_2, and r_2 *inferior* to r_1. This expresses that r_1 may override r_2. For example, given the defeasible rules

```
r:  professor(X) =>  tenured(X)
r': visiting(X)  => ¬tenured(X)
```

which contradict one another, no conclusive decision can be made about whether a visiting professor is tenured. But if we introduce a superiority relation > with r' > r, then we can indeed conclude that a visiting professor is not tenured.

The system works roughly in the following way: to prove a conclusion A defeasibly, there must be a firing rule with A as its head (that is, all literals in the rule body have already been proved); in addition, we must rebut all attacking rules with head the (strong) negation of A. For each such attacking rule we must establish either (a) that this rule cannot fire because we have already established that one of the literals in its body cannot be proved defeasibly (finite failure), or (b) that there is a firing rule with head A superior to the attacking rule.

A formal definition of the proof theory is found in [6]. A model theoretic semantics is found in [36].

3.3. Negation as Failure

Although strong negation is more powerful than negation as failure, the latter is still useful for the Web [44]. For example, consider an auction where we would like to find the best bidder, i.e. the one that offered the

maximum bid. If we assume that bids are represented by facts in the form `bid(X,Y)`, where `X` is the name of the bidder and `Y` is the amount he/she offered, then the following rule finds the maximum bidder and the maximum bid:

`bid(X,Y), not( bid(X1,Y1), X1≠X, Y1>Y ) => max_bid(X,Y)`

Without negation as failure multiple rules would be required in order to find the maximum bidder.

We follow a technique based on auxiliary predicates first presented in [6], but which is often used in logic programming, e.g. [31]. According to this technique, a defeasible theory with NAF can be modularly transformed into an equivalent one without NAF. Every rule

$r: L_1,\ldots,L_n, \text{ not } M_1,\ldots, \text{ not } M_k \Rightarrow L$

can be replaced by the rules:

$r: \; L_1,\ldots,L_n, pr \Rightarrow L$
$\Rightarrow pr$
$\sim L_1 \Rightarrow \neg pr$
$\ldots$
$\sim L_n \Rightarrow \neg pr$

where `pr` is a new propositional atom. If we restrict attention to the original language, the set of conclusions remains the same. In section 6.5 we present a slightly different approach that is used in DR-DEVICE due to the presence of arguments in literals.

3.4. Ambiguity Blocking and Ambiguity Propagation Behavior

A literal is *ambiguous* if there is a chain of reasoning that supports a conclusion that *p* is true, another that supports that ¬*p* is true, and the superiority relation does not resolve this conflict. We can illustrate the concept of ambiguity propagation through the following example.

r_1: quaker(X) => pacifist(X)
r_2: republican(X) => ¬pacifist(X)
r_3: pacifist(X) => ¬hasGun(X)
r_4: livesInChicago(X) => hasGun(X)
quaker(a)
republican(a)
livesInChicago(a)
$r_3 > r_4$

Here `pacifist(a)` is ambiguous. The question is whether this ambiguity should be propagated to the dependent literal `hasGun(a)`. In one defeasible logic variant it is detected that rule r_3 cannot fire, so rule r_4 is unopposed and gives the defeasible conclusion `hasGun(a)`. This behavior is called *ambiguity blocking*, since the ambiguity of `pacifist(a)` has been used to block r_3 and resulted in the unambiguous conclusion `hasGun(a)`. On the other hand, in the *ambiguity propagating* variant, `pacifist(a)` is deemed ambiguous so possibly provable, thus rule r_3 is not recognized as being blocked, so rule r_4 cannot fire unopposed to give the conclusion `hasGun(a)`.

This question has been extensively studied in artificial intelligence, and in particular in the theory of inheritance networks. A preference for ambiguity blocking or ambiguity propagating behavior is one of the properties of nonmonotonic inheritance nets over which intuitions can clash [43]. Ambiguity propagation re-

sults in fewer conclusions being drawn, which might make it preferable when the cost of an incorrect conclusion is high. For these reasons an ambiguity propagating variant of DL is of interest.

3.5. Conflicting Literals

So far we have discussed only conflicts among rules with complementary heads. Namely, we considered all rules with head `L` as *supportive* of `L`, and all rules with head `¬L` as *conflicting*. However, in applications often literals are considered to be conflicting, and at most one of a certain set should be derived. For example, the risk an investor is willing to accept may be classified in one of the categories low, medium, and high. The way to solve this problem is to use constraint rules of the form

```
conflict :: low, medium
conflict :: low, high
conflict :: medium, high
```

Now if we try to derive the conclusion `high`, the conflicting rules are not just those with head `¬high`, but also those with head `low` and `medium`. Similarly, if we are trying to prove `¬high`, the supportive rules include those with head `low` or `medium`.

Another example with arguments in literals would be price negotiation, where an offer should be made by the buyer. The offer can be determined by several rules, whose conditions may or may not be mutually exclusive. All rules have `offer(X)` in their head, since an offer is usually a positive literal. However, only one offer should be made; therefore, only one of the rules should prevail, based on superiority relations among them. In this case, the conflict set is determined as follows:

$C(offer(x)) = \{ \neg offer(x) \} \cup \{ offer(y) \mid y \neq x \}$

In general, given a `conflict :: L, M`, we augment the defeasible theory by:

r_i': $q_1, q_2, \ldots, q_n \rightarrow \neg L$, $\forall r_i$: $q_1, q_2, \ldots, q_n \rightarrow M$
r_i': $q_1, q_2, \ldots, q_n \rightarrow \neg M$, $\forall r_i$: $q_1, q_2, \ldots, q_n \rightarrow L$
r_i': $q_1, q_2, \ldots, q_n \Rightarrow \neg L$, $\forall r_i$: $q_1, q_2, \ldots, q_n \Rightarrow M$
r_i': $q_1, q_2, \ldots, q_n \Rightarrow \neg M$, $\forall r_i$: $q_1, q_2, \ldots, q_n \Rightarrow L$

The superiority relation among the rules of the defeasible theory is propagated to the "new" rules. For example, if the defeasible theory includes the following two rules and a superiority relation among them:

r_1: $q_1, q_2, \ldots, q_n \Rightarrow L$
r_2: $p_1, p_2, \ldots, p_n \Rightarrow M$
$r_1 > r_2$

we will augment the defeasible theory by:

r_1': $q_1, q_2, \ldots, q_n \Rightarrow \neg M$
r_2': $p_1, p_2, \ldots, p_n \Rightarrow \neg L$
$r_1 > r_2'$
$r_1' > r_2$

In section 6.4 we present a generalisation that is used in DR-DEVICE due to the presence of arguments in literals.

3.6. Embedding Defeasible Logic into Logic Programming

Defeasible Logic can be embedded into logic programs through the well-studied meta-program of [35], [7]. Figure 1 shows the meta-program for the ambiguity blocking version of defeasible logic, without taking

into account conflicting literals. Clauses m1 and m2 capture definite provability, which only uses facts and strict rules. Clauses m3-m6 capture defeasible provability. One way of proving a conclusion defeasibly is to prove it definitely (clause m3). Otherwise, to prove X one needs a supportive rule R with head X that fires; in addition, the negation of X must not be definitely provable, and R must not be overruled (clause m4). A supporting rule R is overruled by a not inferior rule S with head the complement of the head of X (clause m5), such that S is not defeated. And a rule S is defeated if there exists an applicable rule, stronger than S and with a complementary literal as its head (m6). In other words, a supporting applicable rule R prevails to prove X defeasibly if every attacking not inferior rule S can be counterattacked by a stronger rule.

```
m1:   definitely(X) :- fact(X).
m2:   definitely(X) :-
          strict(R,X,[Y1,...,Yn]),
          definitely(Y1), ..., definitely(Yn).
m3:   defeasibly(X) :- definitely(X).
m4:   defeasibly(X) :-
          not definitely(~X),
          supportive_rule(R,X,[Y1,...,Yn]),
          defeasibly(Y1) ,..., defeasibly(Yn),
          not overruled(R,X).
m5:   overruled(R,X) :-
          rule(S,~X,[U1,...,Un]),
          defeasibly(U1), ..., defeasibly(Un),
          not defeated(S, ~X).
m6:   defeated(S,~X) :-
          sup(T,S),
          supportive rule(T,X,[V1,...,Vn]),
          defeasibly(V1),...,defeasibly(Vn).

c1:   supportive_rule(Name,Head,Body):- strict(Name,Head,Body).
c2:   supportive_rule(Name,Head,Body):- defeasible(Name,Head,Body).
c3:   rule(Name,Head,Body):- supportive_rule(Name,Head,Body).
c4:   rule(Name,Head,Body):- defeater(Name,Head,Body).
```

Figure 1. The logic meta-program for embedding defeasible reasoning into logic.

4. DR-DEVICE System Architecture

The DR-DEVICE system consists of two major components (Figure 2): the RDF loader/translator and the rule loader/translator. The functionality of the system consists of the following steps:

- The user submits (1) to the rule loader a rule program (a URL or a local file name) that contains:
  - One or more rules in RuleML-like syntax [17].
  - The URL(s) of the RDF input document(s), which is forwarded to the RDF loader.
  - The names of the derived classes to be exported as results.
  - The name of RDF output document.
- The RuleML file is transformed (2) into the native CLIPS-like syntax (3) using the Xalan XSLT processor [46] and an XSLT stylesheet. The DR-DEVICE rule program is then forwarded to the rule translator (4).

- The RDF loader downloads the input RDF documents (5), including their schemas, and it translates RDF descriptions into CLIPS objects (6), according to the RDF-to-object translation scheme of R-DEVICE [13].

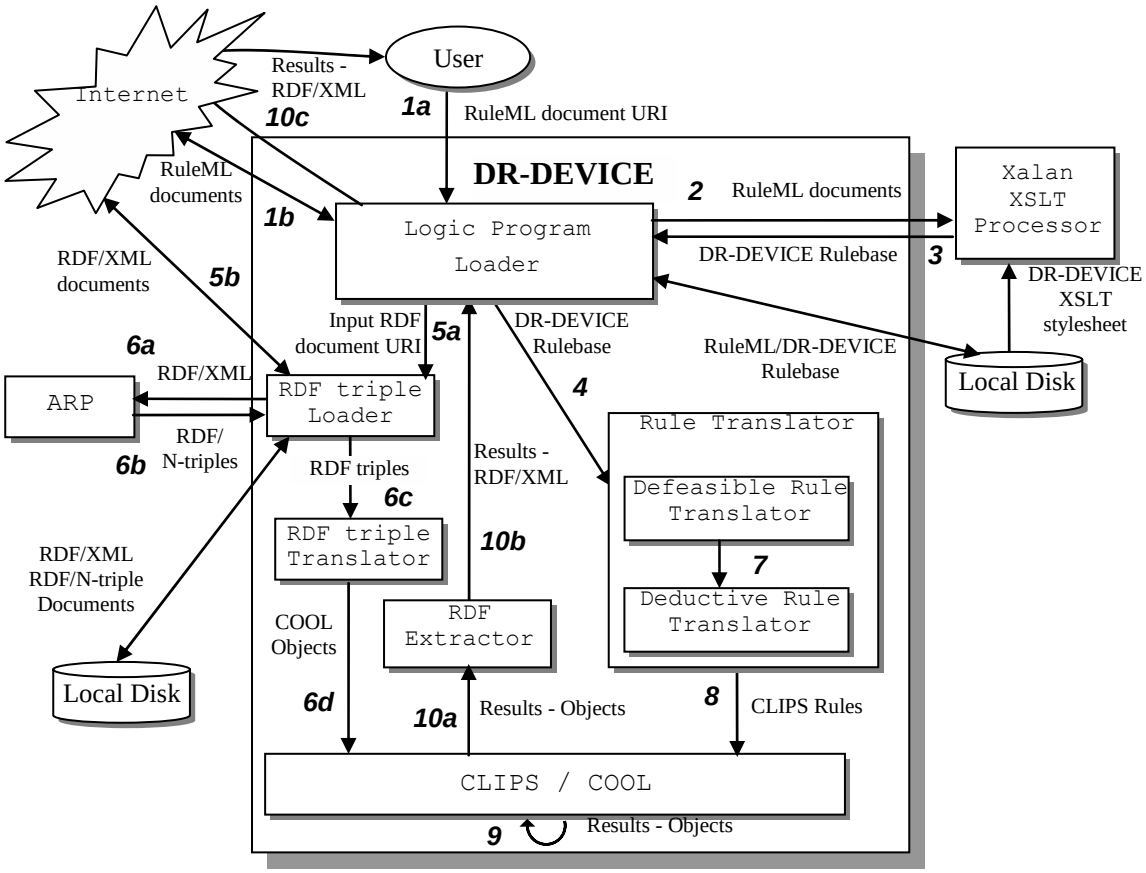

Figure 2. Architecture of the DR-DEVICE system.

- The rule translator compiles the defeasible logic rules into a set of CLIPS production rules. The translation is performed in two steps:
  - First, the defeasible logic rules are translated (7) into sets of deductive, derived attribute and aggregate attribute rules of the basic R-DEVICE rule language, using the translation scheme that is described in section 0.
  - Then, all these R-DEVICE rules are translated (8) into CLIPS production rules, according to the R-DEVICE rule translation scheme [13]. All compiled rule formats are kept into local files, so that the next time they are needed they can be directly loaded, increasing speed.
- The inference engine of CLIPS performs the reasoning by running the production rules and generates the objects that constitute the result of the initial rule program or query (9). The compilation phase guarantees correctness of the reasoning process according to the operational semantics of defeasible logic.
- Finally, the result-objects are exported (10) to the user as an RDF/XML document through the RDF extractor. The RDF document includes:
  - The RDF Schema definitions for the exported derived classes.

- Those instances of the exported derived classes, which have been proven, either positively or negatively, either defeasibly or definitely.

4.1. The R-DEVICE System

In this subsection we give a brief overview of the R-DEVICE system which is the basis for building DR-DEVICE. R-DEVICE is a deductive object-oriented knowledge base system, which transforms RDF triples into objects [12] and uses a deductive rule language [13] for querying and reasoning about them. R-DEVICE imports RDF data into the CLIPS production rule system [19] as COOL objects. The main difference between the established RDF triple-based data model and our OO model is that we treat properties both as first-class objects and as normal encapsulated attributes of resource objects. In this way properties of resources are not scattered across several triples as in most other RDF querying/inferencing systems, resulting in increased query performance due to less joins.

4.1.1. The RDF-to-object Mapping

The main features of this mapping scheme are the following:

- Resource *classes* are represented both as COOL classes and as direct or indirect instances of the `rdfs:Class` class. This binary representation is due to the fact that COOL does not support meta-classes.
- All *resources* are represented as COOL objects, direct or indirect instances of the `rdfs:Resource` class.
- Finally, *properties* are instances of the class `rdf:Property`. Furthermore, properties are defined as slots (attributes) of their domain class(es). The values of properties are stored inside resource objects as slot values.

For example, consider the RDF document in Figure 20 (section 8) which describes available apartments for renting. The RDF Schema for this document lies at the URL of `carlo` namespace (Figure 3). The definition of the class `carlo:apartment` in COOL is shown in Figure 4, along with the corresponding "meta-class" object, which is an instance of the `rdfs:Class`. Notice that the properties that have class `carlo:apartment` 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 `carlo:price` in Figure 5). Finally, Figure 6 shows the COOL object that corresponds to the apartment of Figure 20. Notice that properties on the left column correspond either to RDF properties or RDF-related auxiliary properties, whereas properties on the right column correspond to auxiliary attributes that relate to the defeasible logic translation scheme (section 6).

The descriptive semantics of RDF data may call for dynamic redefinitions of the OO schema, which are effectively handled by R-DEVICE. One example for such a re-definition is when a new property is defined for an existing class. Then the class schema needs to be re-defined in order to include the new property as a slot of the existing class.

The RDF-to-object mapping is performed by the RDF triple loader which accepts from the Rule Loader (or directly from the user) requests for loading specific RDF documents (Figure 2, step 5a). The RDF documents are downloaded from the Internet (5b) and the ARP parser [39] is used (6a) to translate RDF/XML de-

scriptions to triples in the N-triple format (6b). Both the RDF/XML and N-triple files are stored locally for future reference. Furthermore, each RDF document is recursively scanned for namespaces which are also parsed using the ARP parser. The rationale for translating namespaces is to obtain a complete RDF Schema in order to minimize the number of OO schema redefinitions. Fetching multiple RDF schema files will aggregate multiple RDF-to-OO schema translations into a single OO schema redefinition. Namespace resolution is not guaranteed to yield an RDF schema document; therefore, if the namespace URI is not an RDF document, then the ARP parser will not produce triples and DR-DEVICE will make assumptions, based on the RDF semantics [29], about non-resolved properties, resources, classes, etc.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY carlo "http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;" xmlns:xsd="&xsd;"
          xmlns:carlo="&carlo;" >
  <rdfs:Class rdf:about="&carlo;apartment" rdfs:label="apartment">
    <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
  </rdfs:Class>
  <rdf:Property rdf:about="&carlo;bedrooms" rdfs:label="bedrooms">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;central" rdfs:label="central">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;floor" rdfs:label="floor">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;gardenSize" rdfs:label="gardenSize">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;lift" rdfs:label="lift">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;name" rdfs:label="name">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;pets" rdfs:label="pets">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;price" rdfs:label="price">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;size" rdfs:label="size">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
  </rdf:Property>
</rdf:RDF>
```

Figure 3. RDF Schema definition for the class `carlo:apartment`.

N-triples are loaded into memory, while the resources that have a `URI#anchorID` or `URI/anchorID` format are transformed into a `ns:anchorID` format if `URI` belongs to the initially collected namespaces, in

order to save memory space. The transformed RDF triples are fed to the RDF triple translator which maps them into COOL objects and then deletes them. The loading/translation of N-Triples can be performed in either a single step or in an iterative (streaming) fashion where at each iteration only a (user-defined) fragment of the total triples is loaded/translated.

```
(defclass carlo:apartment
  (is-a rdfs:Resource)
  (multislot carlo:size (type INTEGER))
  (multislot carlo:price (type INTEGER))
  (multislot carlo:pets (type LEXEME))
  (multislot carlo:name (type LEXEME))
  (multislot carlo:lift (type LEXEME))
  (multislot carlo:gardenSize
             (type INTEGER))
  (multislot carlo:floor (type INTEGER))
  (multislot carlo:central (type LEXEME))
  (multislot carlo:bedrooms
             (type INTEGER)))
```

```
[carlo:apartment] of rdfs:Class
(uri carlo:apartment)
(source rdf)
(rdfs:isDefinedBy)
(rdf:type [rdfs:Class])
(rdf:value)
(rdfs:comment)
(rdfs:label "apartment")
(rdfs:seeAlso)
(rdfs:subClassOf [rdfs:Resource])
(class-refs rdfs:isDefinedBy rdfs:Resource
            rdf:type rdfs:Class
            rdfs:seeAlso rdfs:Resource)
(aliases rdfs:seeAlso rdfs:isDefinedBy)
```

Figure 4. COOL class definition for the `carlo:apartment` class and corresponding instance of `rdfs:Class`.

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

Figure 5. Instance of `rdf:Property` for the `carlo:price` property.

```
[carlo_ex:a1] of carlo:apartment
(uri carlo_ex:a1)                    (positive 2)
(source rdf)                         (negative 0)
(rdfs:isDefinedBy)                   (positive-derivator nil)
(rdf:type [carlo:apartment])         (negative-derivator nil)
(rdf:value)                          (positive-support)
(rdfs:comment)                       (negative-support)
(rdfs:label "a1")                    (positive-overruled)
(rdfs:seeAlso)                       (negative-overruled)
(carlo:size 50)                      (positive-defeated)
(carlo:price 300)                    (negative-defeated)
(carlo:pets "yes")
(carlo:name "a1")
(carlo:lift "no")
(carlo:gardenSize 0)
(carlo:floor 1)
(carlo:central "yes")
(carlo:bedrooms 1)
```

Figure 6. COOL object for the apartment of Figure 20.

4.1.2. The Rule Language of R-DEVICE

R-DEVICE has a powerful deductive rule language which includes features such as normal (ground), unground, and generalized path expressions over the objects, stratified negation, aggregate, grouping, and

sorting, functions. The rule language supports a second-order syntax, where variables can range over classes and properties. However, second-order variables are compiled away into sets of first-order rules, using instantiations of the metaclasses. Users can define views which are materialized and, optionally, incrementally maintained by translating deductive rules into CLIPS production rules. Users can choose between an OPS5/CLIPS-like or a RuleML-like syntax. Finally, users can use and define functions using the CLIPS host language. R-DEVICE belongs to a family of previous such deductive object-oriented rule languages ([15], [14]). Examples of rules are given below, as well as in [42].

R-DEVICE, like DR-DEVICE, has both a native CLIPS-like syntax and a RuleML-compatible syntax. Here we will present a few examples using the former, since it is more concise. For example, assume that in addition to the RDF Schema of Figure 3 that defines an apartment class for the case study of section 8, there is another class `carlo:owner` that defines the owners of the apartments and a property `carlo:has-owner` that relates an apartment to its owner (Figure 7).

```
  <rdfs:Class rdf:about="&carlo;owner"/>
  <rdf:Property rdf:about="&carlo;has-owner">
    <rdfs:domain rdf:resource="&carlo;apartment"/>
    <rdfs:range rdf:resource="&carlo;owner"/>
  </rdf:Property>
  <rdf:Property rdf:about="&carlo;lastName">
    <rdfs:domain rdf:resource="&carlo;owner"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
```

Figure 7. Additional RDF classes and properties for the RDF Schema of Figure 3.

The following rule returns the names of all apartments owned by "Smith":

```
(deductiverule test1
  (carlo:apartment (carlo:name ?x) ((carlo:lastName carlo:has-owner) "Smith"))
 =>
  (result (apartment ?x))
)
```

The above rule has a ground path expression `(carlo:lastName carlo:has-owner)` where the right-most slot name (`carlo:has-owner`) is a slot of the "departing" class `carlo:apartment`. Moving to the left, slots be-long to classes that represent the range of the predecessor slots. In this example, the range of `carlo:has-owner` is `carlo:owner`, so the next slot `carlo:lastName` has domain `carlo:owner`. The value expression in the above pattern (e.g. constant `"Smith"`) actually describes a value of the left-most slot of the path (`carlo:lastName`). Notice that we have adopted a right-to-left order of attributes, contrary to the left-to-right C-like dot notation that is commonly assumed, because we consider path expressions as function compositions, if we assume that each property is a function that maps its domain to its range.

Another example that demonstrates aggregate function in R-DEVICE is the following rule, which returns the number of apartments owned by each owner:

```
(deductiverule test2
  (carlo:apartment (carlo:name ?x) ((carlo:lastName carlo:has-owner) ?o))
 =>
  (result (owner ?o) (apartments (count ?x)))
)
```

Function `count` is an aggregate function that returns the number of all the different instantiations of the variable `?x` for each different instantiation of the variable `?o`. There are several other aggregate functions, such as `sum`, `avg`, `list`, etc.

5. The Syntax of the Rule Language of DR-DEVICE

DR-DEVICE has two syntaxes: a native CLIPS-like one and a RuleML-compatible one. In this section we briefly introduce the former, concentrating mostly on the latter. There are three types of rules in DR-DEVICE, closely reflecting defeasible logic: strict rules, defeasible rules, and defeaters. Rule type is declared with keywords `strictrule`, `defeasiblerule`, and `defeater`, respectively. For example, the following rule construct (in CLIPS-like notation) represents the defeasible rule `r4: bird(X) => flies(X).`

```
(defeasiblerule r4
    (bird (name ?X))
  =>
    (flies (name ?X)))
```

Predicates have named arguments, called slots, since they represent CLIPS objects. The same rule is represented in RuleML [17] notation (version 0.85) as follows:

```
<imp>
  <_rlab ruleID="r4" ruletype="defeasiblerule">
    <ind>r4</ind>
  </_rlab>
  <_head>
      <atom>
        <_opr>
          <rel>bird</rel>
        </_opr>
        <_slot name="name">
          <var>X</var>
        </_slot>
      </atom>
  </_head>
  <_body>
    <atom>
      <_opr>
        <rel href="flies"/>
      </_opr>
      <_slot name="name">
        <var>X</var>
      </_slot>
    </atom>
  </_body>
</imp>
```

We have tried to re-use as many features of the "official" RuleML syntax as possible. However, several features of the DR-DEVICE rule language could not be captured by the existing RuleML DTDs (0.85[1]); therefore, we have developed a new DTD (Figure 8) using the modularization scheme of RuleML, extending the Datalog with strong negation and negation as failure DTD. Notice that the DTD in Figure 8 does not autonomously capture the full syntax of DR-DEVICE rules, since it is an extension of an existing "official"

[1] In the future we will upgrade to newer XSD-based versions of RuleML (e.g. 0.87).

DTD (http://www.ruleml.org/0.85/dtd/nafneg/nafnegurdatalog.dtd), which is dynamically included in the DR-DEVICE DTD (see boldface in Figure 8).

```
<!ENTITY % LABELs "IDREFS">
<!ENTITY % CLASSes "NMTOKENS">
<!ATTLIST _rlab  ruleID ID #REQUIRED
                 ruletype (strictrule | defeasiblerule | defeater) #REQUIRED
                 superior %LABELs; #IMPLIED >
<!ELEMENT calc %_calc.cont;>      <!ENTITY % _calc.cont "(function_call+)">
<!ENTITY % _head.content " (calc?, (atom | neg))">
<!ENTITY % _body.content "(atom | neg | and | or)">
<!ENTITY % imp.content "(  (_rlab, ( (_head, _body?) |  (_body?, _head) )) |
                           (_head, ( (_rlab, _body?) | (_body, _rlab?) )) |
                           (_body?, ( (_rlab, _head) | (_head, _rlab?) ))    )">
<!ENTITY % naf.content "(atom|and)">
<!ENTITY % pos_term "(ind | var | function_call)">
<!ELEMENT function_call ((%pos_term;)*)>
<!ATTLIST function_call   name CDATA #REQUIRED >
<!ENTITY % term "(_not | %pos_term;)">
<!ELEMENT _not (ind | var)>
<!ELEMENT _or (%term;, (%term;)+)>      <!ELEMENT _and (%term;, (%term;)+)>
<!ENTITY % constraint "(_not | _or | _and)">
<!ENTITY % _slot.content "(ind | var | %constraint;)">
<!ENTITY % rulebase.content "(((_rbaselab, (imp | fact | query | competing_rules)*) |
                            ((imp | fact | query | competing_rules)+, _rbaselab?))?)">
<!ELEMENT _crlab (ind)>
<!ENTITY % competing_rules.content "(_crlab)">
<!ELEMENT competing_rules %competing_rules.content;>
<!ATTLIST competing_rules    c_rules IDREFS #REQUIRED
                             slotnames NMTOKENS #IMPLIED >
<!ENTITY % nafnegurdatalog_include SYSTEM
                             "http://www.ruleml.org/0.85/dtd/nafneg/nafnegurdatalog.dtd">
%nafnegurdatalog_include;
<!ATTLIST rulebase   rdf_import %URI; #IMPLIED
                     rdf_export_classes %CLASSes; #IMPLIED
                     rdf_export CDATA #IMPLIED >
```

Figure 8. DTD for the RuleML syntax of the DR-DEVICE rule language.

For example, rules have a unique (ID) ruleID attribute in their _rlab element, so that superiority of one rule over the other can be expressed through an IDREF attribute of the superior rule. For example, the following rule r5 is superior to rule r4 that has been presented above.

```
(defeasiblerule r5
    (declare (superior r4))
    (penguin (name ?X))
  =>
    (not (flies (name ?X))))
```

In RuleML notation, there is a superiority attribute in the rule label.

```
<imp>
  <_rlab ruleID="r5" ruletype="defeasiblerule" superior="r4">
    <ind>r5</ind>
  </_rlab>
...
</imp>
```

Although there are extensions to RuleML that represent the superiority relation between two rules, external to rules [24], we have adopted this more encapsulated representation scheme that leads to a more efficient translation scheme. However, it is not difficult to implement (at the RuleML level) an external superiority relation and then during translation to encapsulate this relation inside the superior rule. Furthermore,

even using our scheme it is easy to discover superior and inferior rules using appropriate XPATH expressions. For example, the expression `//imp/_rlab[@ruleID="r2"]/@superior` returns all the rules that are inferior to rule `r2`, while the expression `//imp/_rlab[contains(@superior,"r10")]/@ruleID` returns all rules that are superior to rule `r10`.

Strong negation and negation as failure can be expressed in DR-DEVICE rule conditions. Furthermore, strong negation can also occur in the rule head. The strong negation operator is `neg` in the RuleML syntax, and `not` in CLIPS syntax. The negation-as-failure operator is `naf` in both syntaxes.

Classes and objects (facts) can also be declared in DR-DEVICE; however, the focus in this paper is the use of RDF data as facts. The input RDF file(s) are declared in the `rdf_import` attribute of the `rulebase` (root) element of the RuleML document. There exist two more attributes in the `rulebase` element: the `rdf_export` attribute that declares the address of the RDF file with the results of the rule program to be exported, and the `rdf_export_classes` attribute that declares the derived classes whose instances will be exported in RDF/XML format. Further extensions to the RuleML syntax, include function calls that are used either as constraints in the rule body or as new value calculators at the rule head. Furthermore, multiple constraints in the rule body can be expressed through the logical operators: `_not`, `_and`, `_or`. Finally, conflicting literals rules can be declared in DR-DEVICE by a `competing_rules` construct which groups together all rules which compete for a unique positive conclusion. Examples of all these can be found in the section 6.4 (Figure 21, Figure 22).

6. The Translation of DR-DEVICE Rules into R-DEVICE Rules

The translation of defeasible rules into R-DEVICE rules is based on the translation of defeasible theories into logic programs through meta-program of Figure 1. In that program, clauses `c1` to `c4` define the predicates of the classes of rules. Specifically, strict and defeasible rules are considered supportive rules, because they can support the derivation of a conclusion, whereas defeaters can only block the derivation of a conclusion. Furthermore, all rule types are collectively considered as *rules*. These definitions are reflected in the OO schema definition for rule objects of DR-Device, shown in Figure 9.

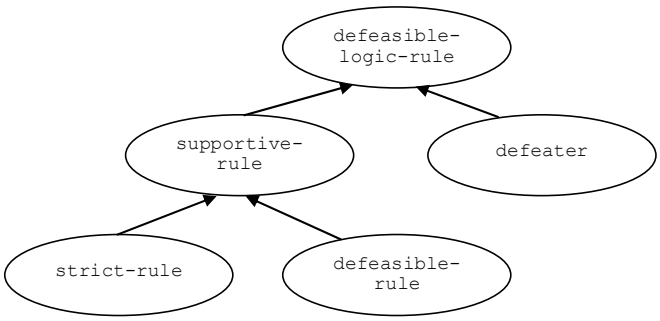

Figure 9. Class hierarchy for DR-DEVICE rules.

The clauses `m1` to `m6` of the meta-program in are not used directly at run-time. Instead they are used to guide defeasible rule compilation. Therefore, at run-time only first-order rules exist. Another big difference

of the meta-program with our translation scheme is that the meta-program expresses the operational semantics of propositional logic, whereas the language of DR-DEVICE supports the use of first-order variables, which range over the values of object attributes, or RDF resource properties, if we consider resources equivalent to objects, as in R-DEVICE.

6.1. Structure of Defeasible Objects

The use of predicate logic complicates things, because each argument in Figure 1 that refers to the head of a rule does not refer to a predicate (i.e. class in our case), which are usually few in a program, but rather must refer to individual ground literals (i.e. specific objects in our case). Therefore, the objects of DR-DEVICE must keep additional auxiliary information (in addition to the user-defined attributes) in order to keep track of this relation to the various rule types that refer to them in their heads. Before going into the details of the translation we briefly present the auxiliary system attributes of each object in DR-DEVICE:

- `positive`, `negative`: These numerical slots hold the proof status of the defeasible object. A value of 1 at the `positive` slot denotes that the object has been defeasibly proven; whereas 2 denotes definite proof. Equivalent values in the `negative` slot denote an equivalent proof status for the negation of the defeasible object. A 0 value for both slots denotes that there has been no proof for either the positive or the negative conclusion (ambiguity). Facts and input RDF meta-data are treated as definitely proved objects, as dictated by clause `m1` in the meta-program of Figure 1.
- `positive-support`, `negative-support`: These attributes hold the rule ids of the rules that can *potentially* prove positively or negatively the object. This means that for each supportive rule in the form:

```
(supportive-rule name
    Condition
  =>
    (Class Slot-Assignments))
```

  all objects of *`Class`* whose attribute values are compatible with *`Slot-Assignments`* have the id *`[name]`* of this rule in their `positive-support` attribute. An equivalent scheme holds for rules that have a negative conclusion.
- `positive-overruled`, `negative-overruled`: These attributes hold the rule ids of the rules that have overruled the positive or the negative proof of the defeasible object. For example, in the rules `r4` and `r5` that were presented in section 5, rule `r5` has a negative conclusion that overrides the positive conclusion of rule `r4`. Therefore, if the condition of rule `r5` is satisfied then its rule id is stored at the `positive-overruled` slot of the corresponding derived object.
- `positive-defeated`, `negative-defeated`: These attributes hold the rule ids of the rules that can defeat overriding rules when the former are superior to the latter. For example, rule `r5` is superior to rule `r4`. Therefore, if the condition of rule `r5` is satisfied then its rule id is stored at the `negative-defeated` slot of the corresponding derived object along with the rule id of the defeated rule `r4`. Then, even if the condition of rule `r4` is satisfied, it cannot overrule the negative conclusion derived by rule `r5` (as it is suggested by the previous paragraph) because it has been defeated by a superior rule.

For example, assume that there is a penguin (also a bird) named Tweety and the rules `r4` and `r5`. Tweety does not fly since it is a penguin. Therefore, while rule `r4` alone concludes that Tweety flies, rule `r5` refutes it. Since `r5` is superior to `r4` the conclusion for the ability of Tweety's ability to fly should be negative. Figure 10 shows the COOL object `[fliestweety]` that corresponds to this conclusion. Notice that the `negative` slot is 1, while the `positive` slot is 0, which means that the conclusion has been defeasibly proven negatively. Furthermore, the `negative-derivator` slot points to rule `r5`, because this conclusion is due to this rule. Slots `positive-support` and `negative-support` point to rules r4 and r5, respectively, because these two rules could potentially prove the positive or negative conclusion. The slot `positive-overruled` contains the pair `r5-overruled r4`, because the positive conclusion of rule `r4` has been overruled by the rule `r5-overruled`, since `r5` is a superior rule (see next sub-section). Finally, the slot `positive-defeated` contains the pair `r5-defeated r4`, because the positive conclusion of rule `r4` has been defeated by the rule `r5-defeated`, since rule `r4` and `r5` have contradictory conclusions (see next sub-section).

```
[fliestweety] of flies
(positive 0)                                (positive-overruled r5-overruled r4)
(negative 1)                                (negative-overruled)
(positive-derivator nil)                    (positive-defeated r5-defeated r4)
(negative-derivator r5)                     (negative-defeated)
(positive-support r4)                       (name tweety)
(negative-support r5)
```

Figure 10. Example of a defeasible object.

6.2. Translation Scheme

In this subsection we present how DR-DEVICE rules are translated into R-DEVICE rules, using the defeasible object structure presented above. We first present the translation of defeasible rules, and then we explain how this scheme differs for strict rules and defeaters.

6.2.1. Defeasible Rules

In the following, we assume that defeasible rules are in the following form:

```
(defeasible-rule pos-name
    (declare (superior Inferior-Rules))
    Condition
   =>
    (Class Slot-Assignments))
```

The above rule has a positive conclusion. Rules with a negative conclusion are assumed to be in the following general form:

```
(defeasible-rule neg-name
    (declare (superior Inferior-Rules))
    Condition
   =>
    (not (Class Slot-Assignments)))
```

Each defeasible rule in DR-DEVICE is translated into a set of 5 R-DEVICE rules:

- A *deductive* rule

- A "*support*" rule (aggregate attribute rule)
- A "*defeasibly*" rule (derived attribute rule)
- An "*overruled*" rule (derived attribute rule)
- A "*defeated*" rule (derived attribute rule)

The deductive rule

The *deductive* rule generates a derived defeasible object when the condition of the defeasible rule is met. The proof status slots of the derived objects are initially set to 0.

```
(deductiverule pos-name-deductive
    Condition
   =>
    (Class Slot-Assignments (positive 0) (negative 0)))
```

Notice that the exactly same rule is generated for negative rules, since the negation of the conclusion does not have to do with the existence of the object.

```
(deductiverule neg-name-deductive
    Condition
   =>
    (Class Slot-Assignments (positive 0) (negative 0)))
```

For example, for rule `r5` the following deductive rule is generated:

```
(deductiverule r5-deductive
    (penguin (name ?X))
   =>
    (flies (name ?X) (positive 0) (negative 0)))
```

Rule `r5-deductive` states that if an object of class `penguin` with slot `name` equal to `?X` exists, then create a new object of class `flies` with a slot `name` with value `?X`. The derivation status of the new object (according to defeasible logic) is unknown since both its positive and negative truth status slots are set to 0. Notice that if a `flies` object already exists with the same name, it is not created again. This is ensured by the value-based semantics of the R-DEVICE deductive rules.

At run-time, if the fact of Figure 11 is present at the working memory, then rule `r5-deductive` will generate the defeasible object of Figure 10. However, both its `positive` and `negative` slots would be 0 and all the other slots would have null values.

```
[penguintweety] of penguin
(positive 2)                                (positive-overruled)
(negative 0)                                (negative-overruled)
(positive-support r2)                       (positive-defeated)
(negative-support)                          (negative-defeated)
                                            (animal-name tweety)
```

Figure 11. A sample fact.

The support rule

The "*support*" rule is an aggregate attribute rule that stores in `...-support` slots the rule ids of the rules that can potentially prove positively or negatively an object. In the following, `list` is an aggregate function that just collects values in a list.

```
(aggregateattrule pos-name-support
    Condition
```

```
    ?var <- (Class Slot-Conditions)
  =>
    ?var <- (Class (positive-support (list pos-name))))
```

In the above rule, *Slot-Conditions* are the conditions that correspond to the *Slot-Assignments* of the original rules.

A similar translation occurs for negative rules; the only difference is the name of slot that the rule id is stored.

```
(aggregateattrule neg-name-support
    Condition
    ?var <- (Class Slot-Conditions)
  =>
    ?var <- (Class (negative-support (list neg-name))))
```

For example, for rule `r5` the following "support" rule is generated:

```
(aggregateattrule r5-support
    (penguin (name ?X))
    ?gen23 <- (flies (name ?X))
  =>
    ?gen23 <- (flies (negative-support (list r5))))
```

Rule `r5-support` states that if there is a `penguin` object named `?X`, and there is a `flies` object with the same name, then derive that rule `r5` could potentially support the defeasible negation of the `flies` object (slot `negative-support`).

At run-time, if the object in Figure 11 is present and after the rule `r5-deductive` generates the corresponding `flies` object, rule `r5-support` adds r5 to the `negative-support` multislot (Figure 10).

The defeasibly rule

The "*defeasibly*" rule is a derived attribute rule that defeasibly proves either positively or negatively an object by storing the value of 1 in the `positive` or `negative` slots, if the rule condition has been at least defeasibly proven (*Defeasible-Condition*), if the opposite conclusion has not been definitely proven `(negative ~2)` and if the rule has not been overruled by another rule (`positive-overruled` condition). This is the exact semantics of the clause `m4` of the meta-program in Figure 1.

```
(derivedattrule pos-name-defeasibly
    Defeasible-Condition
    ?var <- (Class Slot-Conditions
                   (negative ~2)
                   (positive-overruled $?x&:(not (member$ pos-name $?x))))
  =>
    ?var <- (Class (positive 1)))
```

In the above rule, *Defeasible-Condition* is a transformation of the original *Condition*, where each object condition element is required to be defeasibly proven. An object is defeasibly proven (or not proven) if its `positive` (or `negative`) slot has a value of at least 1; this includes objects that are definitely proven (or not proven), whose `positive` (or `negative`) slot equals 2. Here, we actually make use of the clause `m3` of the meta-program in Figure 1. Specifically, each positive condition element $(Class_i\ Slot\text{-}Conditions_i)$ is transformed to:

$(Class_i\ Slot\text{-}Conditions_i$ `(positive ?ps&:(>= ?ps 1)))`

whereas each negative condition element `(not` $(Class_j\ Slot\text{-}Conditions_j)$`)` is transformed to:

```
(Class_j Slot-Conditions_j (negative ?ns&:(>= ?ns 1)))
```

A similar translation occurs for negative rules; the only difference being the names of the slots involved, which are exactly the opposite ones from the positive rule.

```
(derivedattrule neg-name-defeasibly
    Defeasible-Condition
    ?var <- (Class Slot-Conditions
                    (positive ~2)
                    (negative-overruled $?x&:(not (member$ neg-name $?x))))
  =>
    ?var <- (Class (negative 1)))
```

For example, for rule `r5` the following "defeasibly" rule is generated:

```
(derivedattrule r5-defeasibly
  (penguin (name ?X) (positive ?gen29&:(>= ?gen29 1)))
  ?gen23 <- (flies (name ?X) (positive ~2)
                   (negative-overruled $?gen25&:(not (member$ r5 $?gen25))))
 =>
  ?gen23 <- (flies (negative 1)))
```

Rule `r5-defeasibly` states that if it has been defeasibly proven that a `penguin` object named `?X` exists, and there is a `flies` object with the same name that is not already strictly-positively proven and rule `r5` has not been overruled (check slot `negative-overruled`), then derive that the `flies` object is defeasibly-negatively proven.

At run-time, if the object in Figure 11 is present and the rule `r5-deductive` generates the corresponding `flies` object and rule r5 has not been overruled by a stronger positive rule, since `negative-overruled` slot is empty in Figure 10, then rule `r5-defeasibly` changes the value of slot `negative` to 1.

The overruled rule

The "*overruled*" rule is a derived attribute rule that stores in `...-overruled` slots the rule id (*`pos-name`*`-overruled`) of the rule that has overruled the positive or the negative proof of the defeasible object, along with the ids of the rules that support the opposite conclusion `(negative-support $?x1)`, if the rule condition has been at least defeasibly proven (*`Defeasible-Condition`*), and if the rule has not been defeated by a superior rule (`positive-defeated` condition). This is the exact semantics of the clause `m5` of the meta-program in Figure 1.

```
(derivedattrule pos-name-overruled
    Defeasible-Condition
    ?var <- (Class Slot-Conditions
                    (negative-support $?x1)
                    (negative-overruled $?x2)
                    (positive-defeated $?x3&:(not (member$ pos-name $?x3))))
  =>
    (calc (bind $?x4 (create$ pos-name-overruled $?x1 $?x2)))
    ?var <- (Class (negative-overruled $?x4)))
```

In the above rule notice the `calc` expression, through which arbitrary user-defined calculations (through the functional language of CLIPS) are performed in R-DEVICE. In this case, the rule ids of the `negative-support` slot (`$?x1` variable) are concatenated with the rule ids already stored in the `negative-overruled` slot of the object (`$?x2` variable) and the result (`$?x4` variable) is stored back at the `negative-overruled` slot.

A similar translation occurs for negative rules; the only difference being the names of the slots involved, which are exactly the opposite ones from the positive rule.

```
(derivedattrule neg-name-overruled
    Defeasible-Condition
    ?var <- (Class Slot-Conditions
                    (positive-support $?x1)
                    (positive-overruled $?x2)
                    (negative-defeated $?x3&:(not (member$ neg-name $?x3))))
  =>
    (calc (bind $?x4 (create$ neg-name-overruled $?x1 $?x2)))
    ?var <- (Class (positive-overruled $?x4)))
```

For example, for rule `r4` the following "overruled" rule is generated:

```
(derivedattrule r5-overruled
  (penguin (name ?X ) (positive ?gen29 &:(>= ?gen29 1)))
  ?gen23 <- (flies (name ?X) (positive-support $?gen26)
                   (positive-overruled $?gen27)
                   (negative-defeated $?gen25&:(not (member$ r5 $?gen25))))
=>
  (calc (bind $?gen28 (create$ r5-overruled $?gen26 $?gen27)))
  ?gen23 <- (flies (positive-overruled $?gen28)))
```

Rule `r5-overruled` actually overrules all rules that can support the positive derivation of `flies`, including rule `r4`. Specifically, it states that if it has been defeasibly proven that a `penguin` object named `?X` exists, and there is a `flies` object with the same name that can be potentially positively supported by rules in the slot `positive-support` (such as rule `r4`), then derive that rule `r5-overruled` overruled those "positive supporters" (slot `positive-overruled`), unless it has been defeated (check slot `negative-defeated`).

At run-time, if the object in Figure 11 is present and the rule `r5-deductive` generates the corresponding `flies` object and this object can be positively supported by rule `r4` (Figure 10) and rule r5 has not been defeated, since `negative-defeated` slot is empty in Figure 10, then rule `r5-overruled` adds the pair `r5-overruled r4` to the multislot `positive-overruled` (see Figure 10).

The defeated rule

The "*defeated*" rule is a derived attribute rule that stores in `...-defeated` slots the rule id (*`neg-name`*`-defeated`) of the rule that has defeated overriding rules (along with the defeated rule ids - *`Inferior-Rules`*) when the former is superior to the latter and if the rule condition has been at least defeasibly proven (*`Defeasible-Condition`*). A "defeated" rule is generated only for rules that have a superiority relation, i.e. they are superior to others. This is the exact semantics of the clause `m6` of the meta-program in Figure 1.

```
(derivedattrule pos-name-defeated
    Defeasible-Condition
    ?var <- (Class Slot-Conditions (negative-defeated $?x1))
  =>
    (calc (bind $?x2 (create$ pos-name-defeated Inferior-Rules $?x1)))
    ?var <- (Class (negative-defeated $?x2)))
```

In the above rule the `calc` expression concatenates the rules ids of the inferior rules (*`Inferior-Rules`*) with the rule ids already stored in the `negative-defeated` slot of the object (`$?x1` variable) and the result (`$?x2` variable) is stored back at the `negative-defeated` slot.

A similar translation occurs for negative rules; the only difference being the names of the slots involved, which are exactly the opposite ones from the positive rule.

```
(derivedattrule neg-name-defeated
    Defeasible-Condition
    ?var <- (Class Slot-Conditions (positive-defeated $?x1))
  =>
    (calc (bind $?x2 (create$ neg-name-defeated Inferior-Rules $?x1)))
    ?var <- (Class (positive-defeated $?x2)))
```

For example, for rule `r5` the following "defeated" rule is generated:

```
(derivedattrule r5-defeated
    (penguin (name ?X) (positive ?gen29&:(>= ?gen29 1)))
    ?gen23 <- (flies (name ?X) (positive-defeated $?gen26))
=>
    (calc (bind $?gen25 (create$ r5-defeated r4 $?gen26)))
    ?gen23 <- (flies (positive-defeated $?gen25)))
```

Rule `r5-defeated` actually defeats rule `r4`, since `r5` is superior to `r4`. Specifically, it states that if it has been defeasibly proven that a `penguin` object named `?X` exists, and there is a `flies` object with the same name then derive that rule `r5-defeated` defeats rule `r4` (slot `positive-defeated`).

At run-time, if the object in Figure 11 is present and the rule `r5-deductive` generates the corresponding `flies` object, then rule `r5-defeated` adds the pair `r5-defeated r4` to the multislot `positive-defeated` (see Figure 10).

6.2.2. Strict rules

Strict rules are handled in the same way as defeasible rules, with an addition of a derived attribute rule (called *definitely* rule) that definitely proves either positively or negatively an object by storing the value of 2 in the `positive` or `negative` slots, if the condition of the strict rule has been definitely proven (*`Definite-Condition`*), and if the opposite conclusion has not been definitely proven (`negative ~2`). This is the exact semantics of the clause `m2` of the meta-program in Figure 1.

Consider the following general form for a strict rule:

```
(strict-rule pos-name
    Condition
  =>
    (Class Slot-Assignments))
```

The "definitely" rule for the above general form of strict rule is the following:

```
(derivedattrule pos-name-definitely
    Definite-Condition
    ?var <- (Class Slot-Conditions (negative ~2))
  =>
    ?var <- (Class (positive 2)))
```

In the above rule, *`Definite-Condition`* is a transformation of the original *`Condition`*, where each object condition element is required to be definitely proven. An object is definitely proven (or not proven) if its `positive` (or `negative`) slot has exactly value 2. Specifically, each positive condition element (*$Class_i$ $Slot\text{-}Conditions_i$*) is transformed to:

(*$Class_i$ $Slot\text{-}Conditions_i$* `(positive 2)`)

whereas each negative condition element `(not (`*$Class_j$ $Slot\text{-}Conditions_j$*`))` is transformed to:

(*$Class_j$ $Slot\text{-}Conditions_j$* `(negative 2)`)

A similar translation occurs for negative rules; the only difference being the names of the slots involved, which are exactly the opposite ones from the positive rule.

```
(derivedattrule neg-name-definitely
    Definite-Condition
    ?var <- (Class Slot-Conditions (positive ~2))
  =>
    ?var <- (Class (negative 2)))
```

For example, for the strict rule `r3`:

```
(strict-rule
    (penguin (name ?X)
  =>
    (bird (name ?X)))
```

the following "definitely" rule is generated:

```
(derivedattrule r3-definitely
    (penguin (name ?X) (positive 2))
    ?gen9 <- (bird (name ?X) (negative ~2))
  =>
    ?gen9 <- (bird (positive 2)))
```

Rule `r3-definitely` states that if it has been definitely proven that a `penguin` object named `?X` exists, and there is a `bird` object with the same name that is not already strictly-negatively proven, then derive that the `bird` object is definitely-positively proven.

At run-time, if the object in Figure 11 is present and the corresponding "deductive" rule `r3-deductive` has generated the corresponding `bird` object (Figure 12), then rule `r3-definitely` changes the value of slot `positive` to 2 (see Figure 12).

```
[birdtweety] of bird
(positive 2)                          (positive-overruled)
(negative 0)                          (negative-overruled)
(positive-support r3)                 (positive-defeated)
(negative-support)                    (negative-defeated)
                                      (animal-name tweety)
```

Figure 12. A strongly proven object.

6.2.3. Defeaters

Defeaters are much weaker rules that can only overrule a conclusion, according to the semantics of clause `m5` of the meta-program in Figure 1. Therefore, for a defeater only the "overruled" rule is created, along with a deductive rule to allow the creation of derived objects, even if their proof status cannot be supported by defeaters.

6.3. Execution Order

The order of execution of all the above rule types is as follows: "deductive", "support", "definitely", "defeated", "overruled", "defeasibly". Moreover, rule priority for stratified defeasible rule programs is determined by stratification. Finally, for non-stratified rule programs rule execution order is not determined. However, in order to ensure the correct result according to the defeasible logic theory for each derived attribute rule of the rule types "definitely", "defeated", "overruled" and "defeasibly" there is an opposite "truth

maintenance" derived attribute rule that undoes (retracts) the conclusion when the condition is no longer met. In this way, even if rules are not executed in the correct order, the correct result will be eventually deduced because conclusions of rules that should have not been executed can be later undone.

The general form for these truth-maintenance rules is the following:

```
(derivedattrule rule-name-...-dot
    ConclusionOfDerivedAttributeRule
    (not ConditionOfDerivedAttributeRule)
  =>
    UndoConclusionOfDerivedAttributeRule)
```

For example, the following rule undoes the "defeasibly" rule of rule `r5` when either the condition of the defeasible rule is no longer defeasibly satisfied, or the opposite conclusion has been definitely proven, or if rule `r5` has been overruled.

```
(derivedattrule r5-defeasibly-dot
    ?gen23 <- (flies (name ?X) (negative 1) (negative-support $? r5 $?))
    (not
      (and  (penguin (name ?X) (positive ?gen29&:(>= ?gen29 1)))
            ?gen23 <- (flies (positive ~2)
                             (negative-overruled $?g&:(not (member$ r5 $?g))))
      )
    )
  =>
    ?gen23 <- (flies (negative 0)))
```

6.4. Conflicting Literals

The mechanism for dealing with conflicting (mutually exclusive) literals, as presented in section 3.5, is incorporated in DR-DEVICE. In addition, the system also implements a mechanism for dealing with predicate arguments in non-propositional theories. In particular, it handles the case where a predicate c may take at most one value (that is, in case it is functional). In this case, the conflict set is `{(c(x),c(d) | x≠d}`.

For example, suppose that there exist two competing rules `r1` and `r2`, where `r2` is superior, i.e. its conclusion prevails over the conclusion of `r1`:

```
r1: a(X) => c(X)
r2: b(X) => c(X)
r2 > r1
```

DR-DEVICE rewrites the above rule set into the following:

```
r1: a(X) => c(X)
r2: b(X) => c(X)
r1_r2: a(X'), b(X), X≠X' => ~c(X)
r2_r1: b(X'), a(X), X≠X' => ~c(X)
r2 > r1_r2
r2_r1 > r1
```

When both rules `r1` and `r2` fire for a different value of `X`, also do rules `r1_r2` and `r2_r1`. However, rule `r2` is superior to rule `r1_r2` and its conclusion is positively proved. Furthermore, rule `r2_r1` blocks (overrules) the conclusion of rule `r1`. Finally, only the conclusion of rule `r2` is proved.

It is easy to see how the above scheme is generalized to multiple competing rules. For each pair of competing rules r_i, r_j, a new pair of rules $r_i\_r_j$, $r_j\_r_i$, is generated with negated heads. The conditions of the new rules contain the conditions of both competing rules, with changed variable names for all variables

shared with the conclusion. The original superiority relations $r_j > r_i$ become $r_j > r_i\_r_j$. Finally, the new rules are compiled using the translation scheme described above in this section. Notice that competing rules are explicitly declared by the user through a `competing_rules` construct (section 5).

6.5. Negation as Failure

DR-DEVICE supports negation as failure by re-writing each rule that contains a `naf` operator into a set of rules that contain only strong negation, by adapting the emulation technique presented in section 3.3 to handle arguments in literals. For simplicity, we assume that there exists a rule `r` with one `naf` operator (although the same method can be extended to several `naf` operators per rule):

```
r: a(X), naf(b(X)) => c(X)
```

DR-DEVICE rewrites the above rule into the following rule set:

```
r: a(X), r_naf(X) => c(X)
r_naf_pos: a(X) => r_naf(X)
r_naf_neg: a(X), b(X) => ~r_naf(X)
```

where `r_naf(X)` is a system-generated literal. The above rule set exhibits the desired behaviour of negation as failure, because if `b(X)` is not satisfied, while `a(X)` is satisfied, then `r_naf(X)` is concluded by rule `r_naf_pos` and `c(X)` is concluded by rule `r`. If `b(X)` is satisfied then rule `r_naf_neg` defeats rule `r_naf_pos` and rule `r` does not conclude `c(X)`, because its condition is not satisfied. The above approach is generalized to handle multiple `naf` operators similarly to the approach of section 3.3.

7. Performance Evaluation

DR-DEVICE has been extensively tested for correctness and performance using a tool that generates scalable test defeasible logic theories that comes with Deimos, a query answering defeasible logic system [37]. Figures 5-10 show the performance of DR-DEVICE in running theories of various types and sizes. Performance has been measured on a Pentium 4 PC (2GHz) with WinXP Professional and 1GB main memory.

The various theory types explore various aspects of defeasible logic operational semantics. For example, `chain` theories of size *n* start with a fact a_0 and continue with a chain of *n* defeasible rules of the form $a_{i-1} \Rightarrow a_i$. A defeasible proof of a_n will use all of the rules and the fact. A variant `chains` uses only strict rules. Moreover, `levels` theories of size *n* consist of a cascade of $2n+2$ disputed conclusions a_i, $i \in [0 .. 2n+1]$. For each *i*, there are rules $\Rightarrow a_i$ and $a_{i+1} \Rightarrow \neg a_i$. For each odd *i* a priority asserts that the latter rule is superior. A final rule $\Rightarrow a_{2n+2}$ gives uncontested support for a_{2n+2}. A defeasible proof of a_0 will use every rule and priority. A variant `levels-` omits the priorities. More details about the various theory types can be found in [37]. Notice that in our performance evaluation the theory sizes refer to the number of R-DEVICE rules generated from our translation scheme described in section 0.

Results (Figures 5-10) show that DR-DEVICE can handle inferences with thousands of rules within few seconds. In almost all cases the inference time per clause is less than few milliseconds. The performance scalability of DR-DEVICE is equivalent to that of Deimos. However, DR-DEVICE offers a broader range of

functionality, namely predicate logic, instead of propositional logic, support for inference over RDF meta-data, RuleML compliance, conflicting literals, negation as failure, etc.

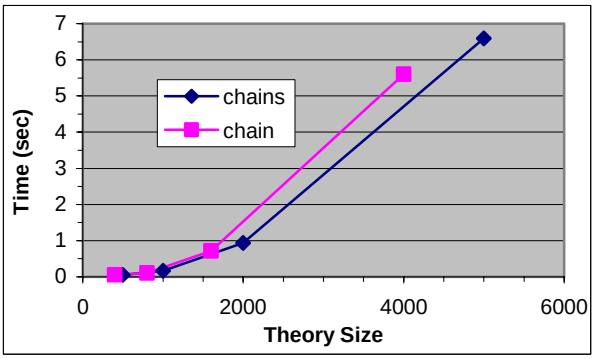

Figure 13. Performance in theories `chain`, `chains`

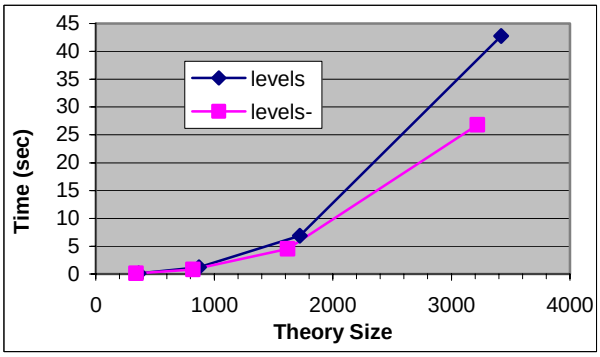

Figure 14. Performance in theories `level`, `levels-`

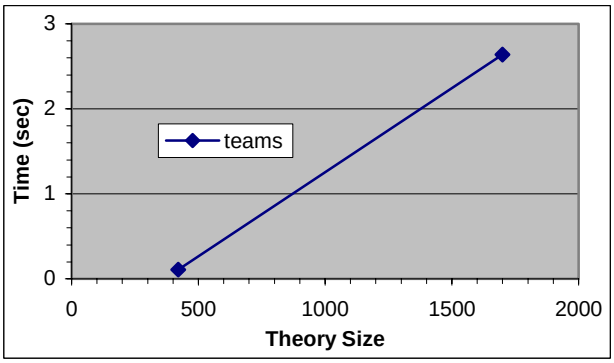

Figure 15. Performance in theory `teams`

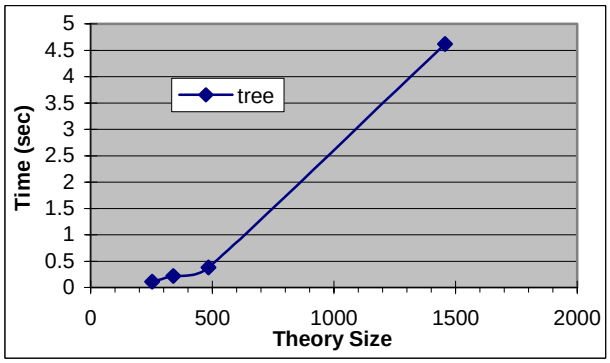

Figure 16. Performance in theory `tree`

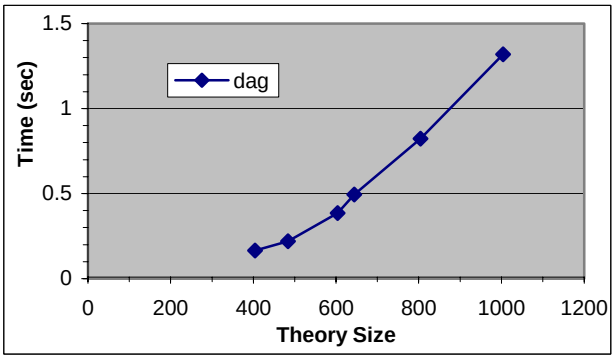

Figure 17. Performance in theory `dag`

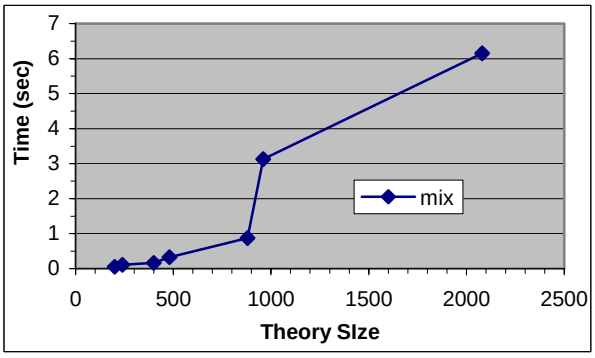

Figure 18. Performance in theory `mix`

8. A Brokered Trade Example

In this section we present a full example of using DR-DEVICE rules in a brokered trade application that takes place via an independent third party, the broker. The broker matches the buyer's requirements and the sellers' capabilities, and proposes a transaction when both parties can be satisfied by the trade. In our case, the concrete application (which has been adopted from [8]) is apartment renting and the landlord takes the role of the abstract seller.

Available apartments reside in an RDF document (Figure 20). The requirements of a potential renter, called e.g. Carlo, are shown in Figure 19. These requirements are expressed in defeasible logic as explained below, in a logic-like syntax. More specifically, the following predicates are used to describe properties of apartments:

- `size(x,y)`, where y is the size of apartment x (in m^2)
- `bedrooms(x,y)`, where apartment x has y bedrooms
- `price(x,y)`, where y is the price for x
- `floor(x,y)`, where apartment x is on the y-th floor
- `gardenSize(x,y)`, where apartment x has a garden of size y
- `lift(x)`, meaning that there is an elevator in the house of x
- `pets(x)`, meaning that pets are allowed in x
- `central(x)`, meaning that x is centrally located

1. *Carlos is looking for an apartment of at least 45m^2 with at least 2 bedrooms. If it is on the 3rd floor or higher, the house must have an elevator. Also, pet animals must be allowed.*
2. *Carlos is willing to pay $300 for a centrally located 45m^2 apartment, and $250 for a similar flat in the suburbs. In addition, he is willing to pay an extra $5 per m^2 for a larger apartment, and $2 per m^2 for a garden.*
3. *He is unable to pay more than $400 in total. If given the choice, he would go for the cheapest option. His 2nd priority is the presence of a garden; lowest priority is additional space.*

Figure 19. Verbal description of Carlo's (a potential renter) requirements.

```
<!DOCTYPE rdf:RDF [
    ...
    <!ENTITY carlo "http://.../dr-device/carlo/carlo.rdf#">
    <!ENTITY carlo_ex "http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo_ex.rdf#">
]>
<rdf:RDF ... xmlns:carlo="&carlo;" xmlns:carlo_ex="&carlo_ex;">
  <carlo:apartment rdf:about="&carlo_ex;a1">
    <carlo:bedrooms rdf:datatype="&xsd;integer">1</carlo:bedrooms>
    <carlo:central>yes</carlo:central>
    <carlo:floor rdf:datatype="&xsd;integer">1</carlo:floor>
    <carlo:gardenSize rdf:datatype="&xsd;integer">0</carlo:gardenSize>
    <carlo:lift>no</carlo:lift>
    <carlo:name>a1</carlo:name>
    <carlo:pets>yes</carlo:pets>
    <carlo:price rdf:datatype="&xsd;integer">300</carlo:price>
    <carlo:size rdf:datatype="&xsd;integer">50</carlo:size>
  </carlo:apartment>
  ...
</rdf:RDF>
```

Figure 20. RDF document for available apartments

Also the following predicates are used:

- `acceptable(x)`, meaning that flat x satisfies Carlos's requirements
- `offer(x,y)`, meaning that Carlos is willing to pay $ y for flat x

Any apartment is a priori acceptable.

```
r1: => acceptable(X)
```

However, Y is unacceptable if one of Carlos's requirements is not met (exceptions to rule `r1`).

```
r2: bedrooms(X,Y), Y < 2 => ¬acceptable(X)
r3: size(X,Y), Y < 45 => ¬acceptable(X)
r4: ¬pets(X) => ¬acceptable(X)
r5: floor(X,Y), Y > 2, ¬lift(X) => ¬acceptable(X)
r6: price(X,Y), Y > 400 => ¬acceptable(X)
r2 > r1, r3 > r1, r4 > r1, r5 > r1, r6 > r1
```

The price Carlos is willing to pay for an apartment is calculated as follows:

```
r7: size(X,Y), Y ≥ 45, garden(X,Z), central(X) => offer(X, 300 + 2Z + 5(Y−45))
r8: size(X,Y), Y ≥ 45, garden(X,Z),¬central(X) => offer(X, 250 + 2Z + 5(Y−45))
```

An apartment is only acceptable if the amount Carlos is willing to pay is not less than the price specified by the landlord.

```
r9: offer(X,Y), price(X,Z), Y < Z => ¬acceptable(X)
r9 > r1
```

In addition to identifying the apartments acceptable to Carlos it is also possible to reduce the number further, even down to a single apartment, by taking further preferences into account. Carlos's preferences are based on price, garden size, and size, in that order, represented as follows:

```
r10: cheapest(X) => rent(X)
r11: cheapest(X), largestGarden(X) => rent(X)
r12: cheapest(X), largestGarden(X), largest(X) => rent(X)
r11 > r10, r12 > r10, r12 > r11
```

Since at most one apartment can be rented, literals `rent(X)` are conflicting. This is represented using conflict sets: `C(rent(x)) = {¬rent(x)} ∪ {rent(y) | y ≠ x}`

The prerequisites of these rules can be derived from the set of acceptable apartments using further rules. For example, `cheapest(X)` can be calculated by the following rule that makes use of negation as failure (operator `not`):

```
rc: acceptable(X), price(X,Z), not(acceptable(Y), Y ≠ X, price(Y,W), W < Z)
        => cheapest(X)
```

Similar rules exist for `largestGarden(X)` and `largest(X)`, as well.

Some of the rules of the very same defeasible logic program in the RuleML-compatible syntax of DR-DEVICE are shown in Figure 21. In the DR-DEVICE version, each apartment is considered a distinct RDF resource (Figure 20) (or object in CLIPS terms) and its properties are RDF properties (or object slots). Things to notice are the following:

- The separate `competing_rules` element to declare that rule `r10`, `r11`, and `r12` are competing, i.e. that their conclusions are conflicting literals.
- The input RDF document and the RDF document that will host the program results are included as attributes of the `rulebase` element, at the beginning of the RuleML document.
- Results include classes `acceptable` and `rent`, meaning that all their instances will be included in the export RDF document.
- The expression of complex constraints on the value of a slot based on logical operators and functions calls, which are directly expressed in XML (rule `r2`).

The complete document can be found in `http://lpis.csd.auth.gr/systems/dr-device.html`.

```
<!DOCTYPE rulebase SYSTEM "http://lpis.csd.auth.gr/systems/dr-device/dr-device.dtd">
<rulebase rdf_import="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#"
          rdf_export_classes="acceptable rent"
          rdf_export="http://lpis.csd.auth.gr/systems/dr-device/carlo/export-carlo.rdf">
  <_rbaselab><ind type="defeasible">carlo-rules</ind></_rbaselab>
  <competing_rules c_rules="r10 r11 r12">
    <_crlab> <ind href="&carlo_rb;cr1">cr1</ind> </_crlab>
  </competing_rules>
  <imp>  <_rlab ruleID="r1" ruletype="defeasiblerule">  <ind href="&carlo_rb;r1">r1</ind>   </_rlab>
         <_head>   <atom>  <_opr>   <rel>acceptable</rel>   </_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="&carlo_rb;r2">r2</ind>   </_rlab>
         <_head><neg>  <atom>   <_opr>   <rel>acceptable</rel>   </_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="r7" ruletype="defeasiblerule"><ind href="&carlo_rb;r7">r7</ind></_rlab>
         <_head>   <calc>  <function_call name="bind">
                              <var>a</var>
                              <function_call name="+">
                                 <ind>300</ind>
                                 <function_call name="*">
                                    <ind>2</ind>
                                    <var>z</var>   </function_call>
                              <function_call name="*">
                                 <ind>5</ind>
                                 <function_call name="-">
                                    <var>y</var>
                                    <ind>45</ind>  </function_call></function_call></function_call>
                           </function_call>  </calc>
                  <atom>  <_opr><rel>offer</rel></_opr>
                           <_slot name="apartment"><var>x</var></_slot>
                           <_slot name="amount"><var>a</var></_slot></atom></_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=">=">
                                          <var>y</var>
                                          <ind>45</ind></function_call></_and></_slot>
                           <_slot name="carlo:gardenSize"><var>z</var></_slot>
                           <_slot name="carlo:central"><ind>"yes"</ind></_slot></atom></_body>
  </imp>
  ...
</rulebase>
```

Figure 21. Part of Carlo's requirements in the RuleML-compatible DR-DEVICE syntax.

After the rule document in Figure 21 is loaded into DR-DEVICE, it is transformed into the native DR-DEVICE syntax (see section 5). DR-DEVICE rules are further translated into R-DEVICE rules, as presented in the previous section, which in turn are translated into CLIPS production rules. All compiled rule formats are kept into local files, so that the next time they are needed they can be directly loaded, increasing speed. Then the RDF document(s) of Figure 20 is loaded and transformed into CLIPS (COOL) objects. Finally, the reasoning can begin, which ends up with 3 acceptable apartments and one suggested apartment for renting, according to Carlo's requirements and the available apartments [8].

The results (i.e. objects of derived classes) are exported in an RDF file according to the specifications posed in the RuleML document (Figure 21). Figure 22 shows an example of the result exported for class `ac-`

`ceptable` (acceptable or not apartments) and class `rent` (suggestions to rent a house or not). Notice that both the positively and negatively proven (defeasibly or definitely) objects are exported. Objects that cannot be at least defeasibly proven, either negatively or positively, are not exported, although they exist inside DR-DEVICE. Furthermore, the RDF schema of the derived classes is also exported.

```
<!DOCTYPE rdf:RDF [ ... <!ENTITY dr-device "http://.../export-carlo.rdf#"> ]>
<rdf:RDF ... xmlns:dr-device='&dr-device;'>
  <rdfs:Class rdf:about='&dr-device;DefeasibleObject'/>
  <rdf:Property rdf:about='&dr-device;truthStatus'>
    <rdfs:domain rdf:resource='&dr-device;DefeasibleObject'/>
    <rdfs:range  rdf:resource='rdfs:Literal'/>
  </rdf:Property>
  <rdfs:Class rdf:about='&dr-device;rent'>
    <rdfs:label rdf:resource='rent'/>
    <rdfs:subClassOf rdf:resource='&dr-device;DefeasibleObject'/>
  </rdfs:Class>
  <rdf:Property rdf:about='&dr-device;apartment'>
    <rdfs:domain rdf:resource='&dr-device;rent'/>
    <rdfs:range  rdf:resource='rdfs:Literal'/>
  </rdf:Property>
...
  <dr-device:acceptable rdf:about="&dr-device;acceptable2">
    <dr-device:apartment>a2</dr-device:apartment>
    <dr-device:truthStatus>defeasibly-not-proven</dr-device:truthStatus>
  </dr-device:acceptable>
  <dr-device:acceptable rdf:about="&dr-device;#acceptable5">
    <dr-device:apartment>a5</dr-device:apartment>
    <dr-device:truthStatus>defeasibly-proven</dr-device:truthStatus>
  </dr-device:acceptable>
...
  <dr-device:rent rdf:about="&dr-device;rent1">
    <dr-device:apartment>a5</dr-device:apartment>
    <dr-device:truthStatus>defeasibly-proven</dr-device:truthStatus>
  </dr-device:rent>
...
</rdf:RDF>
```

Figure 22. Results of defeasible reasoning exported as an RDF document

Concerning the performance of this test case, Table 1 shows the time measured on a Pentium 4 PC (2GHz) with WinXP Professional and 1GB main memory. Times exclude network latencies, i.e. all files are stored locally. Each line shows the time for each task that DR-DEVICE performs. The last line excludes from the total time the execution time of external programs, i.e. the Xalan XSLT processor and the ARP2 RDF parser. When rules are already compiled the time needed to perform the reasoning is 5 times faster (including external programs) or 9 times faster (excluding external programs). In this test case we have included 7 apartments and a total of 15 defeasible rules.

Finally, Figure 23 shows a screenshot from the graphical run-time environment of DR-DEVICE, running the brokered trade example. Specifically, the central window is the main window where the input RuleML file is displayed (can be edited, as well). On the left, the run trace window is shown, whereas on the right the exported results in RDF/XML are shown (in a browser).

Table 1. Performance measurement for the demo

Task	Execution Time (sec)	
	Un-compiled Rules	Compiled Rules
Loading DR-DEVICE system	0.769	0.769
Translating RuleML syntax to DR-DEVICE native syntax (Xalan)	0.934	-
Translating DR-DEVICE rules to R-DEVICE rules	0.604	-
Parsing RDF files (ARP2)	1.868	1.868
Loading RDF into CLIPS	0.220	0.220
Loading R-DEVICE rules	6.648	0.330
Translating R-DEVICE rules	5.659	-
Running R-DEVICE rules	0.275	0.275
Extracting results	~0	~0
Total Execution time	**16.978**	**3.462**
Total DR-DEVICE Execution time	**14.176**	**1.593**

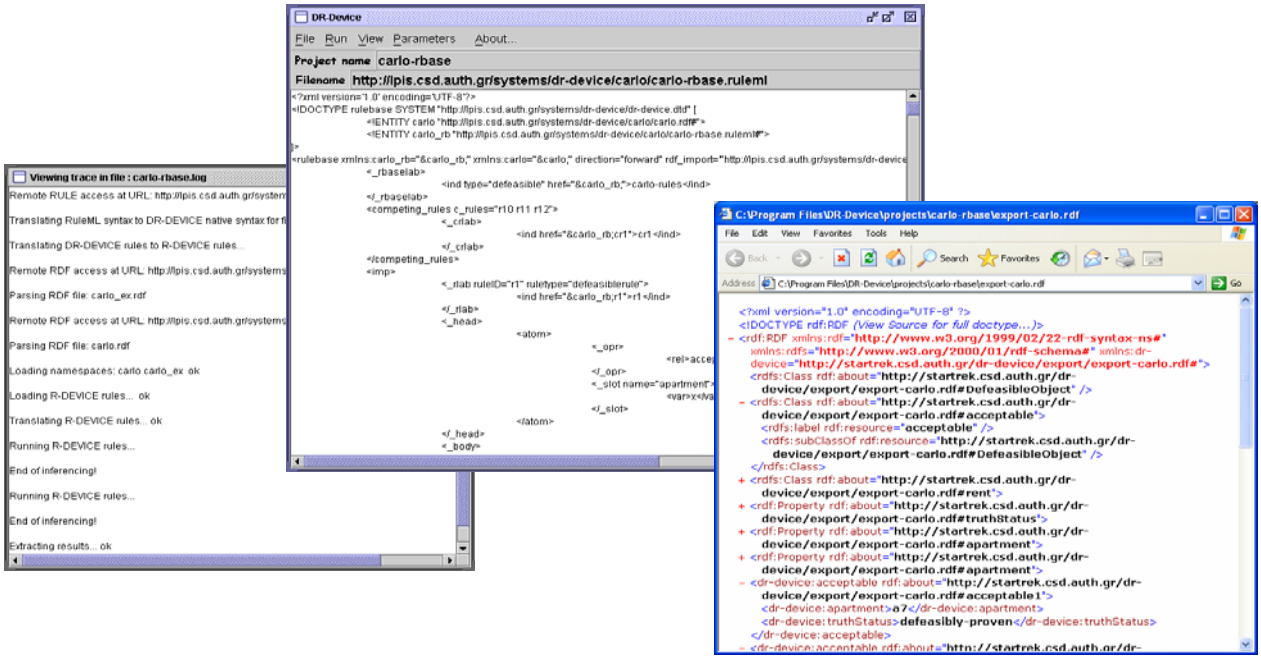

Figure 23. Screenshots of the brokered trade example in DR-DEVICE.

9. Related Work

There exist several previous implementations of defeasible logics. In [21] the historically first implementation, D-Prolog, a Prolog-based implementation is given. It was not declarative in certain aspects (because it did not use a declarative semantic for the not operator), therefore it did not correspond fully to the abstract definition of the logic. Also, D-Prolog supported only one variation thus it lacked the flexibility of the implementation we report on. Finally it did not provide any means of integration with Semantic Web layers and concepts, a central objective of our work.

Deimos [37] is a flexible, query processing system based on Haskell. It implements several variants, but not conflicting literals nor negation as failure in the object language. Also, it does not integrate with Semantic Web (for example, there is no way to treat RDF data and RDFS/OWL ontologies; nor does it use an

XML-based or RDF-based syntax for syntactic interoperability). Thus it is an isolated solution. Finally, it is propositional and does not support variables.

Delores [37] is another implementation, which computes all conclusions from a defeasible theory (the only system of its kind known to us). 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 integrate with other Semantic Web languages and systems, and is thus an isolated solution.

Another Prolog-based implementation of defeasible logics is in [4], which places emphasis on completeness (covering full defeasible logic) and flexibility (covering all important variants). DR-DEVICE is superior in its ability to use many non-logical features of the underlying system CLIPS, thus it is expected to integrate more easily into mainstream IT.

SweetJess [28] is another implementation of a defeasible reasoning system (situated courteous logic programs) based on Jess. It integrates well with RuleML. However, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML-OIL like syntax of RuleML) and not on arbitrary RDF data and ontologies, like DR-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms). This applies to DR-DEVICE to a large extent. However, the basic R-DEVICE language [12] 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. Both of these have been presented in subsection 4.1.2.

Further advantages of DR-DEVICE over SweetJess include the following. SweetJess is more limited in flexibility, in that it implements only one reasoning variant (it corresponds to ambiguity blocking defeasible logic). Finally, DR-DEVICE has a firm formal foundation , as the formal properties of the undelying defeasible logics have been studied extensively and deeply ([6], [7], [9], [34], [35], [36], [37]). These works range from formal properties and formal semantics to correctness proofs for transformations used.

10. Conclusions and Future Work

In this paper we described reasons why conflicts among rules arise naturally on the Semantic Web. To address this problem, we proposed to use defeasible reasoning which is known from the area of knowledge representation. And we reported on the implementation of a system for defeasible reasoning on the Web based on CLIPS production rules. It features:

- Full implementation of defeasible logic, including strict and defeasible rules and defeaters, priorities among rules, conflicting literals, two types of negation and multiple variants regarding ambiguity.
- Therefore reasoning with incomplete and inconsistent information.
- Compatibility with the RuleML syntax and processing of information in the Semantic Web standards of RDF and RDF Schema.
- Efficiency, due to the low computational complexity of the formalism.

The system is freely available for downloading and experimentation (including the test case presented above) at the following address: `http://lpis.csd.auth.gr/systems/dr-device.html`.

Planned future work includes:

- Implementing load/upload functionality in conjunction with an RDF repository, such as RDF Suite [1] and Sesame [18].
- Developing a visual editor for the RuleML-like rule language.
- Deploying the reasoning system as a Web Service.
- Studying in more detail integration of defeasible reasoning with description logic based ontologies. Starting point of this investigation will be the processing of the Horn definable part of OWL [27].
- Applications of defeasible reasoning and the developed implementation for brokering, bargaining, automated agent negotiation, and personalization.

Acknowledgements

This work was partially supported by the European IST Network of Excellence REWERSE and by the PYTHAGORAS II programme which is jointly funded by the Greek Ministry of Education (EPEAEK) and the European Union. The GUI has been developed by Stratos Kontopoulos.

References

[1] Alexaki S., Christophides V., Karvounarakis G., Plexousakis D. and Tolle K., "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases", *Proc. 2nd Int. Workshop on the Semantic Web*, Hong-Kong, 2001.
[2] Antoniou G. and Arief M., "Executable Declarative Business rules and their use in Electronic Commerce", *Proc. ACM Symposium on Applied Computing*, 2002.
[3] Antoniou G., "Nonmonotonic Rule Systems on Top of Ontology Layers", *Proc. 1st Int. Semantic Web Conference*, Springer, LNCS 2342, pp. 394-398, 2002.
[4] Antoniou G., Bikakis A., "A System for Nonmonotonic Rules on the Web", *Submitted*, 2004.
[5] Antoniou G., Billington D. and Maher M.J., "On the analysis of regulations using defeasible rules", *Proc. 32nd Hawaii International Conference on Systems Science*, 1999.
[6] Antoniou G., Billington D., Governatori G. and Maher M.J., "Representation results for defeasible logic", *ACM Trans. on Computational Logic*, 2(2), 2001, pp. 255-287.
[7] Antoniou G., Billington D., Governatori G., Maher M.J, "A Flexible Framework for Defeasible Logics", *Proc. AAAI/IAAI 2000*, AAAI/MIT Press, pp. 405-410.
[8] Antoniou G., Harmelen F. van, *A Semantic Web Primer*, MIT Press, 2004.
[9] Antoniou G., Maher M. J., and Billington D., "Defeasible Logic versus Logic Programming without Negation as Failure", *Journal of Logic Programming*, 42(1), pp. 45-57, 2000.
[10] Antoniou G., Maher M. J., Billington D., "Defeasible Logic versus Logic Programming without Negation as Failure", *Journal of Logic Programming*, 41(1), 2000, pp. 45-57.
[11] Ashri R., Payne T., Marvin D., Surridge M. and Taylor S., "Towards a Semantic Web Security Infrastructure", *Proc. of Semantic Web Services*, 2004 Spring Symposium Series, Stanford University, California, 2004.
[12] Bassiliades N., Vlahavas I., "Capturing RDF Descriptive Semantics in an Object Oriented Knowledge Base System", *Proc. 12th Int. WWW Conf. (WWW2003)*, Budapest, 2003.
[13] Bassiliades N., Vlahavas I., "R-DEVICE: A Deductive RDF Rule Language", *Proc. 3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)*, G. Antoniou, H. Boley (Ed.), Springer-Verlag, LNCS 3323, pp. 65-80, Hiroshima, Japan, 8 Nov. 2004.
[14] Bassiliades N., Vlahavas I., and Sampson D., "Using Logic for Querying XML Data", *Web-Powered Databases*, Ch. 1, pp. 1-35, Idea-Group Publishing, 2003.
[15] Bassiliades N., Vlahavas I., Elmagarmid A.K., "E-DEVICE: An extensible active knowledge base system with multiple rule type support", *IEEE TKDE*, 12(5), pp. 824-844, 2000.
[16] Berners-Lee T., Hendler J., and Lassila O., "The Semantic Web", *Scientific American*, 284(5), 2001, pp. 34-43.

[17] Boley H., Tabet S., *The Rule Markup Initiative*, www.ruleml.org/
[18] Broekstra J., Kampman A. and Harmelen F. van, “Sesame: An Architecture for Storing and Querying RDF Data and Schema Information”, *Spinning the Semantic We*b, Fensel D., Hendler J. A., Lieberman H. and Wahlster W., (Eds.), MIT Press, pp. 197-222, 2003.
[19] *CLIPS Basic Programming Guide* (v. 6.21), www.ghg.net/clips/CLIPS.html
[20] Connolly D., Harmelen F. van, Horrocks I., McGuinness D.L., Patel-Schneider P.F., Stein L.A., DAML+OIL Reference Description, 2001, www.w3c.org/TR/daml+oil-reference
[21] Covington M.A., Nute D., Vellino A., *Prolog Programming in Depth*, 2nd ed., Prentice-Hall, 1997.
[22] Dumas M., Governatori G., Hofstede A. ter and Oaks P., "A formal approach to negotiating agents development", *Electronic Commerce Research and Applications*, 1(2), 2003, pp. 193-207.
[23] Gelder A. van, Ross K. and Schlipf J., “The well-founded semantics for general logic programs”, *Journal of the ACM*, Vol. 38, 1991, pp. 620-650.
[24] Governatori G., "Representing business contracts in RuleML", *International Journal of Cooperative Information Systems*, 14(2-3), pp. 181-216, 2005.
[25] Grosof B. N. and Poon T. C., “SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions”, *Proc. 12th Int. Conf. on World Wide Web*., ACM Press, pp. 340-349, 2003.
[26] Grosof B. N., “Prioritized conflict handing for logic programs”, *Proc. of the 1997 Int. Symposium on Logic Programming*, pp. 197-211, 1997.
[27] Grosof B. N., Horrocks I., Volz R. and Decker S., “Description Logic Programs: Combining Logic Programs with Description Logic”, *Proc. 12th Intl. Conf. on the World Wide Web (WWW-2003)*, ACM Press, 2003, pp. 48-57.
[28] Grosof B.N., Gandhe M.D., Finin T.W., “SweetJess: Translating DAMLRuleML to JESS”, Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML 2002).
[29] Hayes P., “RDF Semantics”, *W3C Recommendation*, Feb. 2004, www.w3c.org/TR/rdf-mt/
[30] Horrocks I., Patel-Schneider P.F., Bechhofer S., Tsarkov D., “OWL Rules: A Proposal and Prototype Implementation”, *Journal of Web Semantics*, (*accepted*), 2005.
[31] Kakas A.C., Mancarella P., and Dung P.M., "The Acceptability Semantics for Logic Programs", *Proc. 11th Int. Conf. on Logic Programming*, pp. 504-519, MIT Press, 1994.
[32] Levy A. and Rousset M.-C., “Combining Horn rules and description logics in CARIN”, *Artificial Intelligence*, Vol. 104, No. 1-2, 1998, pp. 165-209.
[33] Li N., Grosof B. N. and Feigenbaum J.,“Delegation Logic: A Logic-based Approach to Distributed Authorization”, *ACM Trans. on Information Systems Security*, 6(1), 2003.
[34] Maher M. J., "Propositional Defeasible Logic has Linear Complexity", *Logic Programming Theory and Practice*, 1(6), pp. 691-711, 2001.
[35] Maher M. J., Governatori G., "A Semantic Decomposition of Defeasible Logics", *Proc. AAAI’99*, pp. 299-305.
[36] Maher M.J., “A Model-Theoretic Semantics for Defeasible Logic”, *Proc. Workshop on Paraconsistent Computational Logic*, pp. 67-80, 2002.
[37] Maher M.J., Rock A., Antoniou G., Billington D., Miller T., “Efficient Defeasible Reasoning Systems”, *Int. Journal of Tools with Artificial Intelligence*, 10(4), 2001, pp. 483-501.
[38] Marek V.W., Truszczynski M., *Nonmonotonic Logics; Context Dependent Reasoning*, Springer-Verlag, 1993.
[39] McBride B., “Jena: Implementing the RDF Model and Syntax Specification”, *Proc. 2nd Int. Workshop on the Semantic Web*, 2001.
[40] Motik B., Sattler U., Studer R., “Query Answering for OWL-DL with Rules”, *Journal of Web Semantics*, (*accepted*), 2005.
[41] Nute D., “Defeasible logic”, Handbook of logic in artificial intelligence and logic programming (vol. 3): nonmonotonic reasoning and uncertain reasoning, Oxford University Press, 1994.
[42] Seaborne A., and Reggiori A., "RDF Query and Rule languages Use Cases and Examples survey", rdfstore.sourceforge.net/2002/06/24/rdf-query/
[43] Touretzky D.D., Horty J.F. and Thomason R.H., "A Clash of Intuitions: The Current State of Nonmonotonic Inheritance Systems", *Proc. IJCAI-87*, pp. 476-482, Morgan Kaufmann, 1987.
[44] Wagner G., "Web Rules Need Two Kinds of Negation", *Proc. First Workshop on Semantic Web Reasoning*, LNCS 2901, Springer 2003, pp. 33-50.
[45] *Web Ontology Language (OWL)*, http://www.w3c.org/2004/OWL/
[46] *Xalan-Java XSLT processor*, xml.apache.org/xalan-j/