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 briefly presents a semantic web broker example for apartment renting.

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 [37].

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 [26]. Based on that work and on previous work on hybrid reasoning [29] it appears that the best one can do at present is to take the intersection of the expressive power of Horn logic and description logics; one way to view this intersection is the Horn-definable subset of OWL.

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 ([35], [6]) and Courteous Logic Programs [25]. 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 system's main concepts and design have been presented in [11]. 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.
- 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 of RDF ontologies and data as input facts to the defeasible logic program.
- 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 [22].

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 system 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 a use case of a semantic web broker that reasons about apartment renting, using defeasible logic rules. Finally, section 8 briefly overviews related work and section 9 concludes this paper and poses future research directions.

2. Motivation for Conflicting Rules 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 [33].
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 ([10], [30]), business rules [2], personalization, brokering, bargaining, and automated agent negotiations [23].
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 [24] 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 Merging. When ontologies from different authors and/or sources are merged, contradictions arise naturally. Predicate logic based formalisms, including all current Semantic Web languages, cannot cope with inconsistencies. If rule-based ontology languages are used (e.g. DLP [26]) 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. 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

The basic characteristics of defeasible logics are:

- Defeasible logics are rule-based, without disjunction.
- Classical negation is used in the heads and bodies of rules, but negation-as-failure is not used in the object language (it can easily be simulated, if necessary [6], [9]).
- Rules may support conflicting conclusions.
- The logics are skeptical in the sense that conflicting rules do not fire. Thus consistency is preserved.
- Priorities on rules may be used to resolve some conflicts among rules.
- The logics take a pragmatic view and have low computational complexity.

A *defeasible theory D* is a couple $(R,>)$ where R a finite set of rules, and $>$ a superiority relation on R. In expressing the proof theory we consider only propositional rules. Rules containing free variables are interpreted as the set of their variable-free instances.

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: `professor(X) → 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) ⇒ tenured(X)` which reads as follows: “Professors 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 `assistant-Prof(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.

A formal definition of the proof theory is found in [6]. A model theoretic semantics is found in [31].

4. DR-DEVICE System Architecture

The DR-DEVICE system consists of two major components (Fig. 1): the RDF loader/translator and the rule loader/translator. The former accepts from the latter (or the user) requests for loading specific RDF documents. The RDF triple loader downloads the RDF document from the Internet and uses the ARP parser [34] to translate it to triples in the N-triple format. Both the RDF/XML and N-triple files are stored locally for future reference. Furthermore, the 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 [28], about non-resolved properties, resources, classes, etc.

All 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 rule loader accepts from the user a URI (or a local file name) that contains a defeasible logic rule program in RuleML notation [17]. The RuleML document may also contain the URI of the input RDF document on which the rule program will run, which is forwarded to the RDF loader. The RuleML program is translated into the native DR-DEVICE rule notation using the Xalan XSLT processor [38] and an XSLT stylesheet. The DR-DEVICE rule program is then forwarded to the rule translator.

The rule translator accepts from the rule loader (or directly from the user) a set of rules in DR-DEVICE notation and translates them into a set of CLIPS production rules. The translation of the defeasible logic rules is performed in two steps: first, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic R-DEVICE rule language (section 6), and then, all these rules 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. When the translation ends, CLIPS runs the production rules and generates the objects that constitute the result of the initial rule program or query. Finally, the result-objects are exported to the user as an RDF/XML document through the RDF extractor.

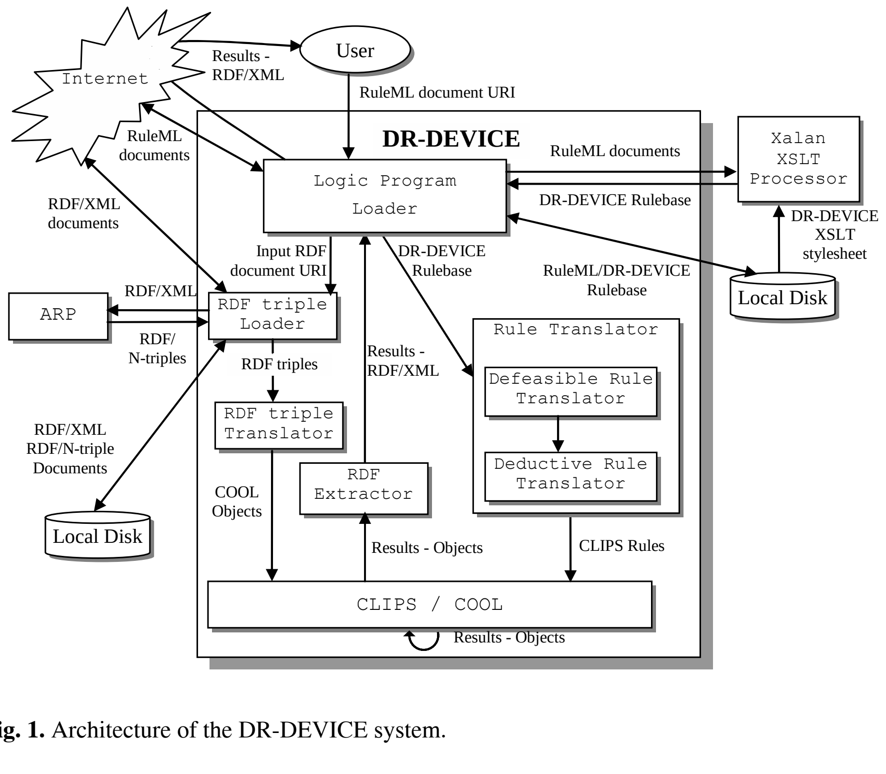

Fig. 1. Architecture of the DR-DEVICE system.

The R-DEVICE system 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. 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.

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 redefinition is when a new property is defined for an existing class.

Furthermore, R-DEVICE features a powerful deductive rule language which is able to express complex inferences both on the RDF schema and data, including recursion, stratified negation, ground and generalized path expressions over the objects, derived attributes and aggregate, grouping, and sorting functions, mainly due to the second-order syntax of the rule language which is efficiently translated into sets of first-order logic rules using metadata. R-DEVICE rules define views which are materialized and, optionally, incrementally maintained. 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]). Deductive rules are implemented as CLIPS production rules and their syntax is a variation of the CLIPS syntax. Examples of rules can be found in the next section, as well as in [36].

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

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 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. DR-DEVICE has also a RuleML-like syntax [17]. The same rule is represented in RuleML 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 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 (**Fig. 2**) using the modularization scheme of RuleML, extending the Datalog with strong negation DTD. 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>
```

```
<!ENTITY % LABELs "IDREFS">                 <!ENTITY % CLASSes "NMTOKENS">
<!ATTLIST _rlab    ruleID ID #REQUIRED
                   ruletype (strictrule | defeasiblerule | defeater) #REQUIRED
                   superior %LABELs; #IMPLIED>
<!ENTITY % _calc.cont "(function+)">        <!ELEMENT calc %_calc.cont;>
<!ENTITY % _head.content " (calc?, (atom | neg))">
<!ENTITY % _body.content "(atom | neg | and | or)">
<!ENTITY % _fname.cont "(#PCDATA)">         <!ELEMENT fname %_fname.cont;>
<!ENTITY % pos_term "(ind | var | function)">
<!ELEMENT function (fname, (%pos_term;)*)>
<!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 % negurdatalog_include SYSTEM
                         " http://www.ruleml.org/0.85/dtd/neg/negurdatalog.dtd">
%negurdatalog_include;
<!ATTLIST rulebase    rdf_import CDATA #IMPLIED
                      rdf_export_classes %CLASSes; #IMPLIED
                      rdf_export CDATA #IMPLIED>
```

Fig. 2. DTD for the RuleML syntax of the DR-DEVICE rule language.

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

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`. Examples of all these can be found in the section 7 (Fig. 6, Fig. 7).

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 the well-studied meta-program of [7]. However, instead of directly using the meta-rpogram at run-time, we have used it to guide defeasible rule compilation. Therefore, at run-time only first-order rules exist.

Before going into the details of the translation we briefly present the auxiliary system attributes (in addition to the user-defined attributes) that each defeasibly derived object in DR-DEVICE has in order to support our translation scheme:

- `pos`, `neg`: These numerical slots hold the proof status of the defeasible object. A value of 1 at the `pos` slot denotes that the object has been defeasibly proven; whereas 2 denotes definite proof. Equivalent `neg` slot values 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.
- `pos-sup`, `neg-sup`: These attributes hold the rule ids of the rules that can *potentially* prove positively or negatively the object.
- `pos-over`, `neg-over`: 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 above, 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 `pos-over` slot of the corresponding derived object.
- `pos-def`, `neg-def`: 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 `neg-def` 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.

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

- A *deductive* rule that generates the 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. For example, for rule r5 the following deductive rule is generated:

```
(deductiverule r5-deductive
  (penguin (name ?X))
 =>
  (flies (name ?X) (pos 0) (neg 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.
- An aggregate attribute "*support*" rule that stores in -sup slots the rule ids of the rules that can potentially prove positively or negatively the object. For example, for rule r5 the following "support" rule is generated (list is an aggregate function that just collects values in a list):

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

  Rule r5-sup 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 neg-sup).
- A derived attribute "*defeasibly*" rule that defeasibly proves either positively or negatively an object by storing the value of 1 in the pos or neg slots, if the rule condition has been at least defeasibly proven, if the opposite conclusion has not been definitely proven and if the rule has not been overruled by another rule. For example, for rule r5 the following "defeasibly" rule is generated:

```
(derivedattrule r5-defeasibly
  (penguin (name ?X) (pos ?gen29&:(>= ?gen29 1)))
  ?gen23 <- (flies (name ?X) (pos ~2)
                   (neg-over $?gen25&:(not (member$ r5 $?gen25))))
 =>
  ?gen23 <- (flies (neg 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 neg-over), then derive that the flies object is defeasibly-negatively proven.
- A derived attribute "*overruled*" rule that stores in -over slots the rule id 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, if the rule condition has been at least defeasibly proven, and if the rule has not been defeated by a superior rule. For example, for rule r4 the following "overruled" rule is generated (through calc expressions, arbitrary user-defined calculations are performed):

```
(derivedattrule r4-over
   (bird (name ?X) (pos ?gen22&:(>= ?gen22 1)))
   ?gen16 <- (flies (name ?X) (neg-sup $?gen19) (neg-over $?gen20)
                    (pos-def $?gen18&:(not (member$ r4 $?gen18))))
 =>
   (calc (bind $?gen21 (create$ r4-over $?gen19 $?gen20)))
   ?gen16 <- (flies (neg-over $?gen21)))
```

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

- A derived attribute "*defeated*" rule that stores in `-def` slots the rule id of the rule that has defeated overriding rules (along with the defeated rule ids) when the former is superior to the latter, if the rule condition has been at least defeasibly proven. A "defeated" rule is generated only for rules that have a superiority relation, i.e. they are superior to others. For example, for rule `r5` the following "defeated" rule is generated:

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

Rule `r5-def` 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-def` defeats rule `r4` (slot `pos-def`).

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 `pos` or `neg` slots, if the condition of the strict rule has been definitely proven, and if the opposite conclusion has not been definitely proven. For example, for the strict rule `r3: penguin(X) → bird(X)`, the following "definitely" rule is generated:

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

Defeaters are much weaker rules that can only overrule a conclusion. 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.

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. 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) (neg 1) (neg-sup $? r5 $?))
   (not (and (penguin (name ?X) (pos ?gen29&:(>= ?gen29 1)))
      ?gen23 <- (flies (pos ~2) (neg-over $?g&:(not (member$ r5 $?g))))))
 =>
   ?gen23 <- (flies (neg 0)))
```

DR-DEVICE has been extensively tested for correctness using a tool that generates scalable test defeasible logic theories that comes with Deimos, a query answering defeasible logic system [32].

7. 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 (Fig. 4). The requirements of a potential renter, called e.g. Carlo, are shown in Fig. 3. These requirements are expressed in DR-DEVICE’s defeasible logic rule language as shown in Fig. 5 (in native CLIPS-like syntax). Rule r2 covers one of the first set of requirements in Fig. 3, rules r7 and r9 represent requirements from the second set and rule r10, from the third. Rule r7 is shown in Fig. 6 in the RuleML-like syntax of DR-DEVICE. Things to notice here is the expression of complex constraints on the value of a slot based on functions calls and logical operators, and the calculation of the values of the slots in the rule head, through again the use of function calls, which are directly expressed in XML.

After the rule document in Fig. 6 is loaded into DR-DEVICE, it is transformed into the native DR-DEVICE syntax (Fig. 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. Then the RDF document(s) of Fig. 4 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].

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.*

Fig. 3. Verbal description of Carlo's (a potential renter) requirements.

```
<!DOCTYPE rdf:RDF [...  <!ENTITY carlo "http://.../dr-device/carlo/carlo.rdf#"> ]>
<rdf:RDF ... xmlns:carlo="&carlo;">
   <carlo:apartment rdf:about="&carlo;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>
```

Fig. 4. RDF document for available apartments

```
 (import-rdf "http://.../dr-device/carlo/carlo.rdf")
(export-rdf "http://.../dr-device/carlo/export-carlo.rdf" acceptable rent)
...
(defeasiblerule r2
   (declare (superior r1))
   (carlo:apartment (carlo:name ?x) (carlo:bedrooms  ?y&:(< ?y 2)))
 =>
   (not (acceptable (apartment ?x))))
...
(defeasiblerule r7
   (carlo:apartment (carlo:name ?x) (carlo:size ?y&:(>= ?y 45))
                      (carlo:gardenSize ?z) (carlo:central "yes"))
 =>
   (calc (bind ?a (+ 300 (* 2 ?z) (* 5 (- ?y 45)))))
   (offer (apartment ?x) (amount ?a)))
...
(defeasiblerule r9
   (declare (superior r1))
   (offer (apartment ?x) (amount ?y))
   (carlo:apartment (carlo:name ?x) (carlo:price ?z&:(< ?y ?z)))
 =>
   (not (acceptable (apartment ?x))))
...
(defeasiblerule r10
   (cheapest (apartment ?x))
 =>
   (rent (apartment ?x)))
...
```

Fig. 5. Part of Carlo's requirements in native (CLIPS-like) DR-DEVICE syntax

```
<!DOCTYPE rulebase SYSTEM "http://.../dr-device/defeasible-dr-device.dtd">
<rulebase rdf_import="http://.../dr-device/carlo/carlo.rdf#"
          rdf_export_classes="acceptable rent"
          rdf_export="http://.../dr-device/carlo/export-carlo.rdf">
  <_rbaselab><ind type="defeasible">carlo-rules</ind></_rbaselab>
  ...
  <imp>
    <_rlab><ind type="defeasiblerule">r7</ind></_rlab>
    <_head> <calc><function><fname>bind</fname>
                            <var>a</var>
                            <function><fname>+</fname>
                                       <ind>300</ind>
                                       <function><fname>*</fname>
                                                  <ind>2</ind>
                                                  <var>z</var>
                                       </function>
                                       <function><fname>*</fname>
                                                  <ind>5</ind>
                                                  <function><fname>-</fname>
                                                             <var>y</var>
                                                             <ind>45</ind>
                                                  </function>
                                       </function>
                            </function>
                 </function>
            </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><fname>>=</fname>
                                                             <var>y</var>
                                                             <ind>45</ind>
                                               </function>
                                            </_and>
                 </_slot>
                 <_slot name="carlo:gardenSize"><var>z</var></_slot>
                 <_slot name="carlo:central"><ind>"yes"</ind></_slot>
            </atom>
    </_body>
  </imp>
  ...
</rulebase>
```

Fig. 6. Part of Carlo's requirements in RuleML-like DR-DEVICE syntax

```
<!DOCTYPE rdf:RDF [ ...
     <!ENTITY dr-device "http://.../dr-device/export/export-carlo.rdf#"> ]>
<rdf:RDF ... xmlns:dr-device='&dr-device;'>
...
  <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: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>
```

Fig. 7. Results of defeasible reasoning exported as an RDF document

The results (i.e. objects of derived classes) are exported in an RDF file according to the specifications posed in the RuleML document (Fig. 6). Fig. 7 shows an example of the result exported for class `acceptable` (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.

8. 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. Finally it did not provide any means of integration with Semantic Web layers and concepts.

Deimos [32] is a flexible, query processing system based on Haskell. It does not integrate with Semantic Web (for example, there is no way to treat RDF data; nor does it use an XML-based or RDF-based syntax). Thus it is an isolated solution.

Delores [32] 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. However, it does not integrate with other Semantic Web languages and systems.

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). However, at present it lacks the ability of processing RDF data.

SweetJess [27] 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, 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. Finally, DR-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules (section 6).

9. 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. It is based on CLIPS production rules, and supports RuleML syntax.

Currently, we are working on extending the rule language with support for negation-as-failure in addition to classical (strong) negation and support for conflicting literals, i.e. derived objects that exclude each other.

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.
- Study in more detail integration of defeasible reasoning with description logic based ontologies. Starting point of this investigation will be the Horn definable part of OWL [26].
- Applications of defeasible reasoning and the developed implementation for brokering, bargaining, automated agent negotiation, and personalization.

10. 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., Billington D., "Defeasible Logic versus Logic Programming without Negation as Failure", *Journal of Logic Programming*, 41(1), 2000, pp. 45-57.

[10] 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.

[11] Bassiliades N., Antoniou G. and Vlahavas I., "DR-DEVICE: A Defeasible Logic System for the Semantic Web", *Proc. 2nd Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR04)*, LNCS 3208, Springer-Verlag, 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", accepted for presentation at *Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)*, 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] 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.

[23] Governatori G., Dumas M., Hofstede A. ter and Oaks P., "A formal approach to legal negotiation", *Proc. ICAIL 2001*, pp. 168-177, 2001.

[24] 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.

[25] Grosof B. N., "Prioritized conflict handing for logic programs", *Proc. of the 1997 Int. Symposium on Logic Programming*, pp. 197-211, 1997.

[26] 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.

[27] 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).*

[28] Hayes P., "RDF Semantics", *W3C Recommendation*, Feb. 2004, `www.w3c.org/TR/rdf-mt/`

[29] 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.

[30] 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.

[31] Maher M.J., "A Model-Theoretic Semantics for Defeasible Logic", *Proc. Workshop on Paraconsistent Computational Logic*, pp. 67-80, 2002.

[32] 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.

[33] Marek V.W., Truszczynski M., *Nonmonotonic Logics; Context Dependent Reasoning*, Springer-Verlag, 1993.

[34] McBride B., "Jena: Implementing the RDF Model and Syntax Specification", *Proc. 2nd Int. Workshop on the Semantic Web*, 2001.

[35] Nute D., "Defeasible logic", *Handbook of logic in artificial intelligence and logic programming (vol. 3): nonmonotonic reasoning and uncertain reasoning*, Oxford University Press, 1994.

[36] Seaborne A., and Reggiori A., "RDF Query and Rule languages Use Cases and Examples survey", `rdfstore.sourceforge.net/2002/06/24/rdf-query/`

[37] *Web Ontology Language (OWL)*, `http://www.w3c.org/2004/OWL/`

[38] *Xalan-Java XSLT processor*, `xml.apache.org/xalan-j/`