Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering 64 (2008) 662–687

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Proof explanation for a nonmonotonic Semantic Web rules language

Grigoris Antoniou [a,*], Antonis Bikakis [a], Nikos Dimaresis [a], Manolis Genetzakis [a], Giannis Georgalis [a], Guido Governatori [b], Efie Karouzaki [a], Nikolas Kazepis [a], Dimitris Kosmadakis [a], Manolis Kritsotakis [a], Giannis Lilis [a], Antonis Papadogiannakis [a], Panagiotis Pediaditis [a], Constantinos Terzakis [a], Rena Theodosaki [a], Dimitris Zeginis [a]

[a] *Institute of Computer Science, FO.R.T.H., Vassilika Vouton, P.O. Box 1385, GR 71110 Heraklion, Greece*
[b] *School of ITEE, The University of Queensland, Australia*

Accepted 17 October 2007
Available online 1 November 2007

Abstract

In this work, we present the design and implementation of a system for proof explanation in the Semantic Web, based on defeasible reasoning. Trust is a vital feature for Semantic Web. If users (humans and agents) are to use and integrate system answers, they must trust them. Thus, systems should be able to explain their actions, sources, and beliefs. Our system produces automatically proof explanations using a popular logic programming system (XSB), by interpreting the output from the proof's trace and converting it into a meaningful representation. It also supports an XML representation for agent communication, which is a common scenario in the Semantic Web. In this paper, we present the design and implementation of the system, a RuleML language extension for the representation of a proof explanation, and we give some examples of the system. The system in essence implements a proof layer for nonmonotonic rules on the Semantic Web.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Defeasible reasoning; Nonmonotonic rule systems; Semantic Web proof layer; Proof explanation; RuleML

1. Introduction

The development of the Semantic Web proceeds in steps, each step building a layer on top of another. At present, the highest layer that has reached sufficient maturity is the ontology layer in the form of the description logic-based language OWL [1]. The next step in the development of the Semantic Web will be the logic and proof layers. The implementation of these two layers will allow the user to state any logical principles, and

[*] Corresponding author.
E-mail address: antoniou@ics.forth.gr (G. Antoniou).

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2007.10.006

permit the computer to infer new knowledge by applying these principles on the existing data. Rule systems appear to lie in the mainstream of such activities.

Many recent studies have focused on the integration of rules and ontologies, and various solutions have been proposed. The Description Logic Programs is the approach followed in [2]; DLPs derive from the intersection of Description Logics and Horn Logic, and enable reasoning with available efficient LP inferencing algorithms over large-scale DL ontologies. We also distinguish the approaches presented in [3,4], which study the integration of Description Logics and Datalog rules. Two representative examples of rule languages for the Semantic Web are TRIPLE [5] and SWRL [6]. They both provide a model for rules on the Semantic Web. TRIPLE is based on F-Logic and provides support for RDFS and a subset of OWL Lite, while SWRL extends OWL DL with Horn-style rules.

Different, but equally interesting research efforts, deal with the standardization of rules for the Semantic Web. Works in this direction include (a) the RuleML Markup Initiative [7], whose ultimate goal is to develop a canonical Web language for rules using XML markup, formal semantics, and efficient implementations; and (b) the research conducted by the Rule Interchange Format (RIF) Working Group, which was recently launched by W3C.

Moreover, rule systems can also be utilized in ontology languages. So, in general rule systems can play a two-fold 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.

Apart from classical rules that lead to monotonic logical systems, recently researchers started to study systems capable of handling conflicts among rules and reasoning with partial information. Recently developed nonmonotonic rule systems for the Semantic Web are:

(a) DR-Prolog [8] is a system that implements the entire framework of Defeasible Logic, and is thus able to reason with: monotonic and nonmonotonic rules, preferences among rules, RDF data and RDFS ontologies. It is syntactically compatible with RuleML, and is implemented by transforming information into Prolog.
(b) DR-DEVICE [9] is also a defeasible reasoning system for the Semantic Web. It is implemented in Jess, and integrates well with RuleML and RDF.
(c) SweetJess [10] implements defeasible reasoning through the use of situated courteous logic programs. It is implemented in Jess, and allows for procedural attachments, a feature not supported by any of the above implementations.
(d) dlvhex [11] is based on *dl-programs*, which realize a transparent integration of rules and ontologies using answer-set semantics.

The upper levels of the Semantic Web have not been researched enough and contain critical issues, like accessibility, trust and credibility. The next step in the architecture of the Semantic Web is the proof layer and little has been written and done for this layer.

The main difference between a query posed to a "traditional" database system and a Semantic Web system is that the answer in the first case is returned from a given collection of data, while for the Semantic Web system the answer is the result of a reasoning process. While in some cases the answer speaks for itself, in other cases the user will not be confident in the answer unless she can trust the reasons why the answer has been produced. In addition it is envisioned that the Semantic Web is a distributed system with disparate sources of information. Thus, a Semantic Web answering system, to gain the trust of a user must be able, if required, to provide an explanation or justification for an answer. Since the answer is the result of a reasoning process, the justification can be given as a derivation of the conclusion with the sources of information for the various steps.

In this work, we describe a system for representing and exchanging explanations on the Semantic Web, which uses Defeasible Logic as the underlying inference system. Defeasible Logic has been shown useful for application areas such as legal reasoning [12], modelling of agents and agent societies [13], agent negotiations [14], semantic brokering [15], and applications to the Semantic Web [8,9].

The paper is organised as follows. Section 2 presents the basics of Defeasible Logic, a nonmonotonic rules system used as the underlying knowledge representation and reasoning method. Section 3 describes the methods used to extract a meaningful explanation from a proof trace. Section 4 outlines a graphical interface used to present an explanation to the user. Section 5 describes the implementation of a multi-agent environment allowing agents to request and receive answers and explanations. Explanations are exchanged in an XML language, an extension of RuleML described in Section 6. Section 7 briefly discusses some potential use cases. Finally, Section 8 discusses related work.

2. Use cases

In this section, we mention three examples where the agents make use of explanations in the Semantic Web:

– An agent can make use of an explanation during a e-commerce negotiation. For example, an agent that represents a buyer can send a message to the agent that represents the online shop asking if the buyer owns money to the shop. If the agent that represents the online shop answers positively, then the buyer's agent may ask for an explanation why he owns the money. Then the online shop's agent will answer sending back the full explanation. This exchange of proofs is crucial for large-scale success of automated e-commerce on the Semantic Web.
– Another case where an agent can use an explanation is at a University System. For example an agent that represents a student may ask for the student's grades. Then for every lesson that the student failed to pass the agent may ask for an explanation why he failed. The university's agent then will respond with a full explanation containing the midterms grade, the grade of the project, and the grade of the final exam. The same may happen for the lessons that the student succeeded; in the latter case, the agents may ask for an explanation about how the grades were extracted. Thus, proofs can enhance the interactive behavior of web sites.
– Finally an agent can ask for an explanation in the case that it is not permitted to access a system. For example, an agent may try to access a system, but the system sends back a message telling that the agent does not have the right permissions to access it. Then the agent may ask for an explanation about why it is not authorized to access the system. An approach in this direction has been developed in the infrastructure described in [16]. That study describes the development of a rule-based management system that provides a mechanism for the exchange of rules and proofs for access control in the Web, in cases such as who owns the copyright to a given piece of information, what privacy rules apply to an exchange of personal information, etc.

In all these cases, the negotiation is made automatically without the user's direct involvement. The agent makes all the appropriate actions and presents only the result and the explanation to the user.

3. Basics of Defeasible Logic

3.1. Basic characteristics

Defeasible reasoning is a simple rule-based approach to reasoning with incomplete and inconsistent information. It represents facts, rules, and priorities among rules. This reasoning family comprises Defeasible Logics [17] and Courteous Logic Programs [18]; the latter can be viewed as a special case of the former [19]. 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. 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 [19]).

– 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.

3.2. Syntax

A *defeasible theory* is a triple $(F, R, >)$, where F is a set of literals (called *facts*), 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 $A \rightarrow p$, where A is a finite set of literals and p is a literal, and are interpreted in the classical sense: whenever the premises are indisputable (e.g. facts) then so is the conclusion. An example of a strict rule is "*Professors are faculty members*". Written formally:

$$professor(X) \rightarrow faculty(X)$$

Inferences from facts and strict rules only are called *definite inferences*. Facts and strict rules are intended to define relationships that are definitional in nature. Thus, Defeasible Logics contain no mechanism for resolving inconsistencies in definite inference.

Defeasible rules are denoted by $A \Rightarrow p$, and can be defeated by contrary evidence. An example of such a rule is

$$professor(X) \Rightarrow tenured(X)$$

which reads as follows:"*Professors are typically tenured*".

Defeaters are denoted by $A \rightsquigarrow p$ and are used to prevent some conclusions. In other words, they are used to defeat some defeasible rules by producing evidence to the contrary. An example is the rule

$$assistant\text{-}professor\ (X) \rightsquigarrow \neg\ tenured\ (X)$$

which reads as follows: "Assistant professors may not be tenured". This means that the information that someone is an assistant professor is not sufficient evidence to conclude that he is not tenured. It is only evidence that he *may* not be tenured.

A superiority relation is an acyclic relation $>$ on R (that is, the transitive closure of $>$ is irreflexive). Given two rules $r1$ and $r2$, if we have that $r1 > r2$, then we will say that $r1$ is *superior* to $r2$, and $r2$ *inferior* to $r1$. This expresses that $r1$ may override $r2$. For example, given the rules

$$r : professor(X) \Rightarrow tenured(X)$$
$$r' : visiting(X) \Rightarrow \neg tenured(X)$$

which contradict each other, 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 he/she cannot be tenured.

3.3. Proof theory

A *conclusion* of D is a tagged literal and can have one of the following four forms:

– $+\Delta q$, which is intended to mean that q is definitely provable in D.
– $-\Delta q$, which is intended to mean that we have proved that q is not definitely provable in D.
– $+\partial q$, which is intended to mean that q is defeasibly provable in D.
– $-\partial q$, which is intended to mean that we have proved that q is not defeasibly provable in D.

If we are able to prove q definitely, then q is also defeasibly provable. This is a direct consequence of the formal definition below. It resembles the situation in, say, default logic: a formula is sceptically provable from a default theory $T = (W, D)$ (in the sense that it is included in each extension) if it is provable from the set of facts W.

Provability is based on the concept of a *derivation* (or *proof*) in $D = (F, R, >)$. A derivation is a finite sequence $P = (P(1),\ldots,P(n))$ of tagged literals constructed by inference rules. There are four inference rules (corresponding to the four kinds of conclusion) that specify how a derivation may be extended. ($P(1\ldots i)$ denotes the initial part of the sequence P of length i):

$+\Delta$: We may append $P(i+1) = +\Delta q$ if either
$q \in F$ or
$\exists r \in R_s[q] \quad \forall a \in A(r) : +\Delta a \epsilon P(1\ldots i)$

That means, to prove $+\Delta q$ we need to establish a proof for q using facts and strict rules only. This is a deduction in the classical sense. No proofs for the negation of q need to be considered (in contrast to defeasible provability below, where opposing chains of reasoning must be taken into account, too).

To prove $-\Delta q$, that is, that q is not definitely provable, q must not be a fact. In addition, we need to establish that every strict rule with head q is *known to be* inapplicable. Thus, for every such rule r there must be at least one antecedent a for which we have established that a is not definitely provable ($-\Delta a$):

$-\Delta$: We may append $P(i+1) = -\Delta q$ if
$q \notin F$ and
$\forall r \in R_s[q] \exists a \in A(r) : -\Delta a \in P(1\ldots i)$

It is worth noticing that this definition of nonprovability does not involve loop detection. Thus, if D consists of the single rule $p \rightarrow p$, we can see that p cannot be proven, but Defeasible Logic is unable to prove $-\Delta p$:

$+\partial$: We may append $P(i+1) = +\partial q$ if either
(1) $+\Delta q \in P(1\ldots i)$ or
(2) (2.1) $\exists r \in R_{sd}[q]\ \forall a \in A(r)$: $+\partial a \in P(1\ldots i)$ and
(2.2) $-\Delta \sim q \in P(1\ldots i)$ and
(2.3) $\forall s \in R[\sim q]$ either
(2.3.1) $\exists a \in A(s)$: $-\partial a \in P(1\ldots i)$, or
(2.3.2) $\exists t \in R_{sd}[q]$ such that
$\forall a \in A(t)$: $+\partial a \in P(1\ldots i)$ and $t > s$

Let us illustrate this definition. To show that q is provable defeasibly we have two choices: (1) we show that q is already definitely provable; or (2) we need to argue using the defeasible part of D as well. In particular, we require that there must be a strict or defeasible rule with head q which can be applied (2.1). But now we need to consider possible attacks, that is, reasoning chains in support of $\sim q$. To be more specific: to prove q defeasibly we must show that $\sim q$ is not definitely provable (2.2). Also (2.3) we must consider the set of all rules which are not known to be inapplicable and which have head $\sim q$. Essentially each such rule s attacks the conclusion q. For q to be provable, each such rule must be counterattacked by a rule t with head q with the following properties: (i) t must be applicable at this point, and (ii) t must be stronger than s. Thus, each attack on the conclusion q must be counterattacked by a stronger rule.

The definition of the proof theory of Defeasible Logic is completed by the condition–∂. It is nothing more than a strong negation of the condition $+\partial$:

$-\partial$: We may append $P(i+1) = -\partial q$ if
(1) $-\Delta q \in P(1\ldots i)$ and
(2) (2.1) $\forall r \in R_{sd}[q]\ \exists a \in A(r)$: $-\partial a \in P(1\ldots i)$ or
(2.2) $+\Delta \sim q \in P(1\ldots i)$ or
(2.3) $\exists s \in R[\sim q]$ such that

(2.3.1) $\forall a \in A(s)$: $+\partial a \in P(1\ldots i)$ and
(2.3.2) $\forall t \in R_{sd}[q]$ either
$\exists a \in A(t)$: $-\partial a \in P(1\ldots i)$ or $t \not> s$

To prove that q is not defeasibly provable, we must first establish that it is not definitely provable. Then we must establish that it cannot be proven using the defeasible part of the theory. There are three possibilities to achieve this: either we have established that none of the (strict and defeasible) rules with head q can be applied (2.1); or $\sim q$ is definitely provable (2.2); or there must be an applicable rule r with head $\sim q$ such that no possibly applicable rule s with head $\sim q$ is superior to s (2.3).

The elements of a derivation P in D are called *lines* of the derivation. We say that a tagged literal L is provable in $D = (F, R, >)$, denoted $D \vdash L$, iff there is a derivation in D such that L is a line of P. When D is obvious from the context we write $\vdash L$.

It is instructive to consider the conditions $+\partial$ and $-\partial$ in the terminology of *teams*, borrowed from Grosof [18]. At some stage there is a team A consisting of the applicable rules with head q, and a team B consisting of the applicable rules with head $\sim q$. These teams compete with one another. Team A wins iff every rule in team B is overruled by a rule in team A; in that case we can prove $+\partial q$. Another case is that team B wins, in which case we can prove $+\partial \sim q$. But there are several intermediate cases, for example one in which we can prove that neither q nor $\sim q$ are provable. And there are cases where nothing can be proved (due to loops).

Concepts of a model-theoretic semantics in [20], and argumentation semantics is discussed in [21].

3.4. Defeasible Logic metaprogram

In order to perform reasoning over a defeasible theory, we have adopted the approach proposed in [22,23]. This approach is based on a translation of a defeasible theory D into a logic metaprogram $P(D)$, and describes a framework for defining different versions of Defeasible Logics, following different intuitions. In particular, this framework is based on two “parameters”:

(a) The metaprogram $P(D)$.
(b) The negation semantics adopted to interpret the negation appearances within $P(D)$.

For part (b), different semantics have been proposed and studied in the literature: the well-founded semantics, answer set semantics, and the classical negation-as-failure operator of Prolog.

In this work, we use a similar metaprogram which fits better our needs for representing explanations. This metaprogram consists of the following program clauses. The first two clauses define the class of rules used in a defeasible theory:

```
supportive_rule(Name,Head,Body):- strict(Name,Head,Body).
supportive_rule(Name,Head,Body):- defeasible(Name,Head,Body).
```

The following clauses define definite provability: a literal is definitely provable if it is a fact or is supported by a strict rule, the premises of which are definitely provable:

```
definitely(X):- fact(X).
definitely(X):- strict(R,X,L),definitely_provable(L).
definitely_provable([X1|X2]):- definitely_provable(X1), definitely_provable(X2).
definitely_provable(X):- definitely(X).
definitely_provable([]).
```

The next clauses define defeasible provability: a literal is defeasibly provable, either if it is definitely provable, or if its complementary is not definitely provable, and the literal is supported by a defeasible rule, the premises of which are defeasibly provable, and which is unblocked.

```
defeasibly(X):- definitely(X).
defeasibly(X):- negation(X,X1), supportive_rule(R,X,L), defeasibly_provable(L),
unblocked(R,X), xsb_meta_not(definitely(X1)).
defeasibly_provable([X1|X2]):- defeasibly_provable(X1),defeasibly_provable(X2).
defeasibly_provable(X):- defeasibly(X).
defeasibly_provable([]).
```

A rule is unblocked when there is not an undefeated conflicting rule:

```
unblocked(R,X):- negation(X,X1), xsb_meta_not(undefeated(X1)).
```

A literal is undefeated when it is supported by a defeasible rule which in not blocked:

```
undefeated(X):- supportive_rule(S,X,L), xsb_meta_not(blocked(S,X)).
```

A rule is blocked either if its premises are not defeasibly provable, or if it is defeated:

```
blocked(R,X):- supportive_rule(R,X,L), xsb_meta_not(defeasibly_provable(L)).
blocked(R,X):- supportive_rule(R,X,L), defeasibly_provable(L), defeated(R,X).
```

A rule is defeated when there is a conflicting defeasible rule, which is superior and its premises are defeasibly provable:

```
defeated(S,X):- negation(X,X1), supportive_rule(T,X1,V),
defeasibly_provable(V), sup(T,S).
```

We define the predicate *negation* to represent the negation of a predicate and evaluate the double negation of a predicate to the predicate itself. Furthermore, we define the predicate *xsb_meta_not* in order to represent the *not* predicate when executing a program in XSB trace.

Our system supports both positive and negative conclusions. Thus, it is able to give justification why a conclusion cannot be reached. We define when a literal is not definitely provable and when it is not defeasibly provable. A literal is not definitely provable if it is not a fact and for every strict rule that supports it, its premises are not definitely provable. A literal is not defeasibly provable if it is not definitely provable and its complementary is definitely provable or for every defeasible rule that supports it, either its premises are not defeasibly provable, or it is not unblocked.

4. Explanation in Defeasible Logic

4.1. Search tree construction

The metaprogram works in conjunction with a Prolog system. In our prototype we use XSB Prolog. It was chosen mainly because we were able to experiment with various LP semantics (usual Prolog not, and well-founded semantics). However, it should be clear that the ideas and functionality of our system are orthogonal to the selection of a particular Prolog system.

The foundation of the proof system lies in the Prolog metaprogram that implements Defeasible Logic, with some additional constructs to facilitate the extraction of traces from XSB. We use the trace of the invocation of the metaprogram to generate a Defeasible Logic search tree, that subsequently will be transformed into a proof suitable to be presented to an end user.

The negation we use in conjunction with the metaprogram is the negation-as-failure of Prolog. Unfortunately, the XSB trace information for well-founded semantics does not provide the information we would require to produce meaningful explanations.

To enable the trace facility, the XSB process executes the command *trace*. After loading of the metaprogram and the defeasible theory, the system is ready to accept any queries which are forwarded unmodified to the XSB process. During the evaluation of the given query/predicate the XSB trace system will print a message each time a predicate is:

1. initially entered (**Call**),
2. successfully returned from (**Exit**),
3. failed back into (**Redo**), and
4. completely failed out of (**Fail**).

The produced trace is incrementally parsed by the Java XSB invoker front-end and a tree whose nodes represent the traced predicates is constructed. Each node encapsulates all the information provided by the trace. Namely:

– A string representation of the predicate's name.
– The predicate's arguments.
– Whether it was found to be true (**Exit**) or false (**Fail**).
– Whether it was failed back into (**Redo**).

In addition to the above, the traced predicate representation node has a Boolean attribute that encodes whether the specific predicate is negated. That was necessary for overcoming the lack of trace information for the *not* predicate (see next section).

One remark is due at this stage: Of course our work relies on the trace provided by the underlying logic programming system (in our case XSB). If we had used an LP directly for knowledge representation, explanations on the basis of LP would have been appropriate. However, here we use defeasible reasoning for knowledge representation purposes (for reasons explained extensively in previous literature), thus explanations must also be at the level of Defeasible Logics.

4.2. Search tree pruning: illustration

The pruning algorithm, that produces the final tree from the initial XSB trace, focuses on two major points. Firstly, the XSB trace produces a tree with information not relevant for the generation of the search tree. One reason for this is that we use a particular metaprogram to translate the Defeasible Logic into logic programming. For the translation to be successful, we need some additional clauses which add additional information to the XSB trace. Another reason derives from the way Prolog evaluates the clauses, showing both successful and unsuccessful paths. Secondly, the tree produced by the XSB trace is built according to the metaprogram structure but the final tree needs to be in a complete different form, compliant with the XML schema described in Section 6. We will take a closer look at the details of these issues.

A main issue of the pruning process was the way Prolog evaluates its rules. Specifically, upon rule evaluation the XSB trace returns all paths followed whether they evaluate to true or false. According to the truth value and the type of the root node, however, we may want to maintain only successful paths, only failed paths or combinations of both. Suppose we have the following defeasibly theory, translated into logic programming clauses as follows:

```
fact(a).
fact(e).
defeasible(r1,b,a).
defeasible(r2,b,e).
defeasible(r3,~(b),d).
```

If we issue a query about the defeasible provability of literal b, XSB trace fails at first to prove that b is definitely provable and then finds a defeasible rule and proves that its premises are defeasible provable. It produces a search tree, which begins with the following lines:

```
Proof
    defeasibly(b):True
        definitely(b):False
            fact(b):False
            strict(_h144,b,_h148):False
        negation(b,~(b)):True
        supportive_rule(r1,b,a):True
            strict(_h155,b,_h157):False
            defeasible(r1,b,a):True
        defeasibly_provable(a):True
            defeasibly(a):True
                definitely(a):True
                    fact(a):True
        ......
```

In this type of proof, we are only interested in successful paths and the pruning algorithm removes the subtree with the false goal to prove that b is definitely provable and the false predicate to find a strict supportive rule for b. It also prunes the metaprogram additional negation clause. The complementary literal is used in next parts of the proof. The corresponding final pruned subtree for this query has the following form:

```
Proof
    defeasibly(b):True
        supportive_rule(r1,b,a):True
            defeasible(r1,b,a):True
        defeasibly_provable(a):True
            defeasibly(a):True
                definitely(a):True
                    fact(a):True
        ......
```

Suppose we have the following defeasibly theory:

```
fact(a).
defeasible(r1,b,a).
defeasible(r2,~(b),a).
```

If we issue a query about the defeasible provability of literal b, XSB fails to prove it; at first it fails to prove that b is definitely provable. It produces a search tree, which begins with the following lines:

```
Proof
    defeasibly(b):False
        definitely(b):False
            fact(b):False
            strict(_h144,b,_h148):False
        ......
```

In this proof, we are interested in unsuccessful paths and the pruning algorithm keeps the initial search tree. Thus, the pruned tree remains the same in the first lines.

The other heuristic rules deal with the recursive structure of Prolog lists and the XSB's caching technique which shows only the first time the whole execution tree for a predicate, during trace execution of a goal. Our pruning algorithm keeps a copy of the initial trace so as to reconstruct the subtree for a predicate whenever it is required.

Using these heuristic techniques, we end up with a version of the search tree that is intuitive and readable. In other words, the tree is very close to an explanation derived by the use of pure Defeasible Logic.

4.3. Search tree pruning: the methods

1. **Pruning Definitely**. When the atom of a query can be proved definitely it is either a fact, in which case we simply locate the fact clause, or there is at least one strict rule having the atom as its head and a body that is definitely provable. Thus, we locate the first such rule along with the definite proof of its body. An example of an initial search tree produced by XSB and the corresponding pruned subtree is the following:

```
Proof
 definitely(b): True
   fact(b): False
   strict(r1, b, a): True
   definitely_provable(a): True
     definitely(a): True
       fact(a): True
```

```
Proof
 definitely(b): True
   strict(r1, b, a): True
   definitely_provable(a): True
     definitely(a): True
       fact(a): True
```

2. **Pruning Not Definitely**. When the atom of a query cannot be proved definitely, it is not a fact and there is no strict rule supporting the atom that its body can be definitely proved. Therefore, we locate the failed 'fact' clause as well as all the aforementioned strict rules along with the proof that their bodies are not definitely provable. An example of an initial search tree produced by XSB and the corresponding pruning subtree is the following:

```
Proof
 definitely(b): False
   fact(b): False
   strict(r2, b, d): True
   definitely_provable(d): False
     definitely(d): False
       fact(d): False
       strict(_h175, d, _h179): False
   strict(_h134, b, _h138): False
```

```
Proof
 definitely(b): False
   fact(b): False
   strict(r2, b, d): True
   definitely_provable(d): False
     definitely(d): False
       fact(d): False
       strict(_h175, d, _h179): False
```

3. **Pruning Defeasibly**. When the atom of a query can be proved defeasibly, it is either definitely provable, in which case we just locate that proof, or there is at least one supportive rule that triggers and is not blocked by any other rule. In the latter case, we locate the first such rule along with the proof of its body as well as the proof that all attacking rules are blocked (either not firing, or defeated). Here we also need to check that the negation of the atom is not definitely provable and we locate that proof as well. An example of an initial search tree produced by XSB and the corresponding pruned subtree is the following (the underlined elements are pruned from the initial search tree):

```
Proof
  defeasibly(b): True
    definitely(b): False
      fact(b): False
      strict(_h144, b, _h148): False
    negation(b, ~(b)): True
    supportive_rule(r1, b, a): True
      strict(_h155, b, _h157): False
      defeasible(r1, b, a): True
    defeasibly_provable(a): True
      defeasibly(a): True
        definitely(a): True
          fact(a): True
    unblocked(r1, b): True
      negation(b, ~(b)): True
      Not supportive_rule(_h286, ~(b), _h288): False
        supportive_rule(r2, ~(b), a): True
          strict(_h286, ~(b), _h288): False
          defeasible(r2, ~(b), a): True
      negation(b, _h267): False
      negation(b, ~(b)): True
        Not undefeated(~(b)): True
         undefeated(~(b)): False
        supportive_rule(r2, ~(b), a): True
          strict(_h566, ~(b), _h564): False
          defeasible(r2, ~(b), a): True
        Not blocked(r2, ~(b)): False
          blocked(r2, ~(b)): True
            supportive_rule(r2, ~(b), a): True
              strict(r2, ~(b), _h886): False
              defeasible(r2, ~(b), a): True
            Not defeasibly_provable(a): False
              defeasibly_provable(a): True
            supportive_rule(r2, ~(b), _h886): False
              defeasible(r2, ~(b), _h886): False
            supportive_rule(r2, ~(b), a): True
              strict(r2, ~(b), _h886): False
              defeasible(r2, ~(b), a): True
            defeasibly_provable(a): True
            defeated(r2, ~(b)): True
              negation(~(b), b): True
              supportive_rule(r1, b, a): True
                strict(_h969, b, _h971): False
                defeasible(r1, b, a): True
              defeasibly_provable(a): True
              sup(r1, r2): True
        supportive_rule(_h566, ~(b), _h564): False
          defeasible(_h566, ~(b), _h564): False
Not definitely(~(b)): True
  definitely(~(b)): False
    fact(~(b)): False
    strict(_h590, ~(b), _h594): False
```

4. **Pruning Not Defeasibly**. When the atom of a query cannot be proved defeasibly, it cannot be proved definitely and either there is a triggering not blocked rule supporting the negation of the atom or there is no triggering supportive rule that is not blocked or the negation of the atom can be definitely proved. An example of an initial search tree produced by XSB and the corresponding pruned subtree is the following:

```
Proof
 defeasibly(b): False
   definitely(b): False
     fact(b): False
     strict(_h144, b, _h148): False
   negation(b, ~(b)): True
   supportive_rule(r1, b, a): True
     strict(_h155, b, _h157): False
     defeasible(r1, b, a): True
   defeasibly_provable(a): True
     defeasibly(a): True
       definitely(a): True
         fact(a): True
   unblocked(r1, b): False
     negation(b, ~(b)): True
  Not supportive_rule(_h286, ~(b), _h288): False
    supportive_rule(r2, ~(b), a): True
      strict(_h286, ~(b), _h288): False
      defeasible(r2, ~(b), a): True
  negation(b, _h267): False
  negation(b, ~(b)): True
  Not undefeated(~(b)): False
    undefeated(~(b)): True
      supportive_rule(r2, ~(b), a): True
        strict(_h566, ~(b), _h564): False
        defeasible(r2, ~(b), a): True
      Not blocked(r2, ~(b)): True
        blocked(r2, ~(b)): False
          supportive_rule(r2, ~(b), a): True
            strict(r2, ~(b), _h886): False
            defeasible(r2, ~(b), a): True
          Not defeasibly_provable(a): False
            defeasibly_provable(a): True
          supportive_rule(r2, ~(b), _h886): False
            defeasible(r2, ~(b), _h886): False
          supportive_rule(r2, ~(b), a): True
            strict(r2, ~(b), _h886): False
            defeasible(r2, ~(b), a): True
          defeasibly_provable(a): True
          defeated(r2, ~(b)): False
            negation(~(b), b): True
            supportive_rule(r1, b, a): True
              strict(_h969, b, _h971): False
              defeasible(r1, b, a): True
            defeasibly_provable(a): True
            sup(r1, r2): False
            defeasibly_provable(a): False
            supportive_rule(_h969, b, _h971): False
              defeasible(_h969, b, _h971): False
            negation(~(b), _h952): False
          defeasibly_provable(a): False
          supportive_rule(r2, ~(b), _h886): False
            defeasible(r2, ~(b), _h886): False
  negation(b, _h267): False
defeasibly_provable(a): False
supportive_rule(_h155, b, _h157): False
  defeasible(_h155, b, _h157): False
negation(b, _h136): False
```

5. **Pruning Lists**. In Prolog, lists have a recursive structure (i.e. a list is a concatenation of an object with the remaining list) and this structure is inherited by the search tree. To remedy this case we flatten the lists to a single depth sequence of atoms. An example of an initial search tree produced by XSB and the corresponding pruned subtree is the following:

```
Proof
  defeasibly(c): True
    definitely(c): True
      fact(c): False
      strict(r1, c, [a, b]): True
      definitely_provable([a, b]): True
        definitely_provable(a): True
          definitely(a): True
            fact(a): True
        definitely_provable([b]): True
          definitely_provable(b): True
            definitely(b): True
              fact(b): True
          definitely_provable([]): True
            definitely([]): False
              fact([]): False
              strict(_h301, [], _h305): False
```

6. **Handling missing proofs**. XSB uses a caching technique in order to avoid reevaluating already evaluated expressions. Effectively, this means that the first time it encounters a predicate, XSB provides the result along with the proof execution tree in the trace. If it comes across the same predicate again, it uses the cached value and does not show the whole execution tree. In some cases, the afore-mentioned pruning techniques may prune the first evaluation of the predicate and at some point where we actually want the predicate to be maintained we are left with the cached version. This is not desirable, so we are forced to keep a copy of the initial trace so as to recover a possibly pruned predicate evaluation subtree. An example of an initial search tree produced by XSB and the corresponding pruned subtree is the following:

```
Proof
 defeasibly(d): True
   definitely(d): True
     fact(d): False
     strict(r3, d, [b, c]): True
     definitely_provable([b, c]): True
       definitely_provable(b): True
         definitely(b): True
           fact(b): False
           strict(r1, b, a): True
           definitely_provable(a): True
             definitely(a): True
               fact(a): True
       definitely_provable([c]): True
         definitely_provable(c): True
           definitely(c): True
             fact(c): False
             strict(r2, c, b): True
             definitely_provable(b): True
         definitely_provable([]): True
           definitely([]): False
```

5. Graphical user interface to the proof system

The graphical user interface Fig. 1 to the proof system, offers an intuitive way to interact with the underlying system and visualize the requested proofs. The proofs are rendered as a tree structure in which each node represents a single predicate. A tree node may have child nodes that represent the simpler, lower level, predicates that are triggered by the evaluation of the parent predicate. Thus, the leaf nodes represent the lowest level predicates of the proof system, which correspond to the basic atoms of a defeasible theory (facts, rules and superiority relations) which cannot be further explained. Additionally, if an atom has additional metadata attached to its definition, such as references for the *fact* and *rule* predicates, those are displayed as a tooltip to the corresponding tree node. This is an optional reference to a Web address that indicates the origin of the atom.

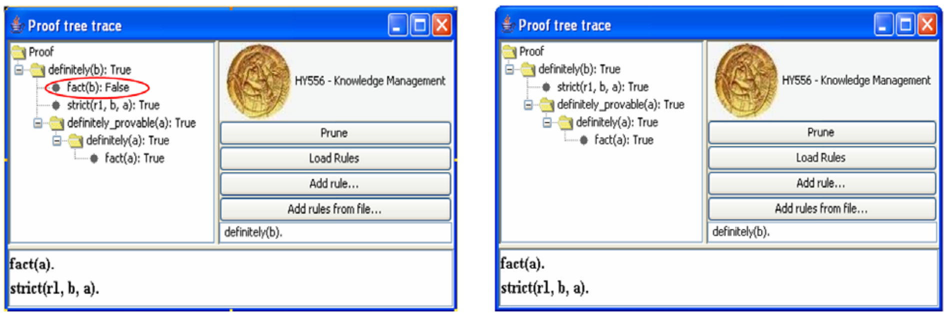
The interaction with the graphical user interface is broken down to three or four steps, depending on whether it is desirable to prune the resulting search tree in order to eliminate the artifacts of the meta-program and simplify its structure (see Section 2) or not. Thus, to extract a proof, the following steps must be carried out:

1. The Defeasible Logic rules must be added to the system. Rules can be added by pressing either the *Add Rule* or *Add rules from file* button at the right part of the interface. The *Add Rule* button presents a text entry dialog where a single rule may be typed by the user. Besides that, pressing the *Add rules from file* button allows the user to select a file that contains multiple rules separated by newlines. The added rules are always visible at the bottom part of the graphical user interface.
2. As soon as the rules are loaded, the system is ready to be queried by the user. By typing a 'query' at the text entry field at the right part of the screen, just below the buttons, and pressing enter, the underlying proof system is invoked with the supplied input and the resulting proof is visualized to the tree view at the left part of the interface.
3. By pressing the *Prune* button the system executes the pruning algorithms described in the previous section to eliminate redundant information and metaprogram artifacts and thus bring the visualized search tree to a more human friendly form.

6. Agent interface to the proof system

The system makes use of two kinds of agents, the 'Agent' which issues queries and the 'Main Agent' which is responsible to answer the queries. Both agents are based on JADE (Java Agent DEvelopment Framework)[24]. JADE simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications. The agent platform can be distributed across machines and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required.

Fig. 2 shows the process followed by the Main Agent in order to answer a query.



```
Proof
  definitely(b): True
    fact(b): False
    strict(r1, b, a): True
    definitely_provable(a): True
      definitely(a): True
        fact(a): True
```

definitely(b).

fact(a).
strict(r1, b, a).

```
Proof
  definitely(b): True
    strict(r1, b, a): True
    definitely_provable(a): True
      definitely(a): True
        fact(a): True
```

definitely(b).

fact(a).
strict(r1, b, a).

Fig. 1. Graphical user interface to the proof system, before and after pruning a proof.

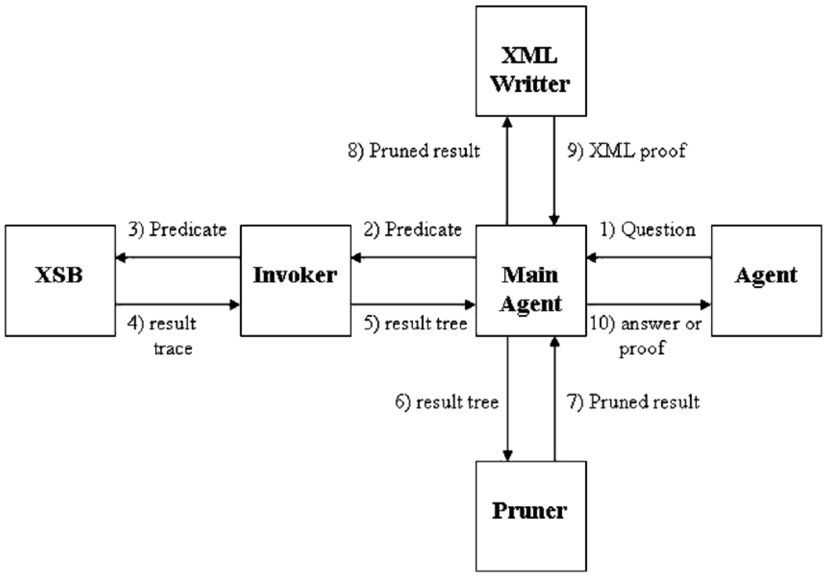


Fig. 2. The system architecture.

All the above steps are illustrated next:

1. **An agent issues a query to the Main Agent**. The query is of the form:**predicate::(proof/answer)**The predicate is a valid Prolog predicate, while the value in the parenthesis indicates the form of the answer that the Main Agent is expected to return. The 'answer' value is used to request for a single true/false answer to the agent's query, while the 'proof' value is used to request for the answer along with its proof explanation. Two examples of queries follow below:
   ```
   defeasibly(rich(antonis))::proof
   defeasibly(rich(antonis))::answer
   ```
2. **Main Agent sends the Predicate to the Invoker**. After receiving a query from an agent, the Main Agent has to execute the predicate. For this reason it extracts the predicate from the query and sends it to the Invoker who is responsible for the communication with the XSB (Prolog engine).
3. **Invoker executes the Predicate**. The Invoker receives the predicate from the Main Agent and sends it to XSB.
4. **XSB returns the result trace**. The XSB executes the predicate and then returns the full trace of the result to the Invoker.
5. **Invoker returns the result tree to Main Agent**. The Invoker receives the trace from the XSB and creates an internal tree representation of it. The result tree is then sent back to the Main Agent.
6. **Main Agent sends the result tree to the Pruner**. The Main Agent after receiving the result tree from the Invoker sends it to the Pruner in order to prune the tree. There exist two kinds of pruning. One is used when the agent that issued the query wants to have the answer only. In that case the tree is pruned and the remaining is just the answer (true or false). The other type of pruning is used when the agent that issued the query also wants to have the proof. In that case, the branches of the tree that are not needed are pruned, so the remaining is a pruned tree containing only the branches that are needed.
7. **Pruner returns the pruned result**. The pruned result is sent back to the Main Agent.
8. **Main Agent sends the pruned result to the XML writer**. This step is used only when the agent that issued the query wants the proof. In this step the pruned result (proof) is sent to the XML writer in order to create an XML representation of the proof.
9. **XML writer returns the XML Proof**. The XML writer creates an XML representation of the proof, according to the XML schema, and sends it back to the Main Agent.

10. **Main Agent returns Answer or Proof**. Finally the Main Agent sends back to the agent that issued the query a string that contains the answer (true, false) or the proof accordingly to what he asked. The format of the string that is sent follows one of the three patterns:
    - **ANSWER(true–false)** e.g ANSWERtrue This pattern is used when the Main Agent wants to send only the answer. In this case it sends the string 'ANSWER' followed by the string representation of the answer (i.e. 'true' or 'false'). There is no space between the two words.
    - **PROOF:(proof string)** This pattern is used when the Main Agent wants to send the proof. In this case it sends the string 'PROOF:' followed by the string representation of the proof (written in XML)
    - **ERROR:(error message)** e.g. ERROR:invalid mode This pattern is used when an error occurs during the process. In this case the Main Agent sends the string 'ERROR:' followed by the string that contains the error message.

7. Extension of RuleML for explanation representation

The need for a formal, XML based, representation of an explanation in the Semantic Web led us to design an extension of the Rule Markup Language (RuleML) [13]. RuleML is an XML based language that supports rule representation for the Semantic Web. In this section, we describe in detail the design of a new XML schema, extension of RuleML, for explanation representation in Defeasible Logic and in the next section we give some instructive examples.

7.1. Atoms, facts and rule representation

In our XML schema, we use a similar syntax to RuleML in order to represent *Facts* and *Rules*. Specifically, we use the *Atom* element which refers to an atomic formula, and it consists of two elements, an operator element (`Op`) and either a Variable element (`Var`) or an Individual constant element (`Ind`), preceded optionally by a not statement (in case we represent a negative literal). Fig. 3 shows the declaration of a typical Atom.

Similarly to RuleML, a *Fact* consists an Atom that comprises a certain knowledge. The last primitive entity of our schema is *Rules*. In Defeasible Logic, we distinguish two types of Rules: *Strict Rules* and *Defeasible Rules*. In our schema we define two different elements for these two types of rules. Both elements consist of two parts, the `Head` element which is constituted of an Atom element, and the `Body` element which is constituted of a number of Atom elements. Fig. 4 shows a typical example of a Defeasible Rule.

7.2. Definitely provable explanations

The simplest proof explanation is the case of a definitely provable Atom. For that proof, we first have to denote the Atom, and then give the `Definite Proof` that explains why it is definitely provable. This explanation can come out in two ways: either a simple `Fact` for that Atom, or a `Strict Rule` with this Atom as its `Head` and an Atom that should be also proved definitely with the same way as its `Body`. If the `Body` consists of multiple Atoms, then we state the definite provable explanation for every atom of the Body. Fig. 5 shows the structure of a definite proof explanation.

```
<Atom>
    <Not>
        <Op>  rich </Op>
        <Ind> Bob  </Ind>
    </Not>
</Atom>
```

Fig. 3. Declaration of an Atom.

```
<Defeasible_rule Label="r1">
    <Head>
        <Atom>
            <Op>  rich </Op>
            <Ind> Bob  </Ind>
        </Atom>
    </Head>
    <Body>
        <Atom>
            <Op>  wins_lotto </Op>
            <Ind> Bob        </Ind>
        </Atom>
    </Body>
</Defeasible_rule>
```

Fig. 4. Declaration of a defeasible rule.

```
<Definitely_provable>
    <Atom>
        <Op>  rich </Op>
        <Ind> Bob  </Ind>
    </Atom>
    <Definite_Proof>
        <Strict_rule Label="r1">
            <Head>
                <Atom>
                    <Op>  rich </Op>
                    <Ind> Bob  </Ind>
                </Atom>
            </Head>
            <Body>
                <Atom>
                    <Op>  wins_lotto </Op>
                    <Ind> Bob        </Ind>
                </Atom>
            </Body>
        </Strict_rule>
        <Definitely_provable>
            <Definite_Proof>
                <Fact>
                    <Atom>
                        <Op>  wins_lotto </Op>
                        <Ind> Bob        </Ind>
                    </Atom>
                </Fact>
            </Definite_Proof>
        </Definitely_provable>
    </Definite_Proof>
</Definitely_provable>
```

Fig. 5. Example of a typical definite proof explanation.

7.3. Defeasibly provable explanations

A defeasibly provable explanation arises from the Defeasible Logic specification. If an Atom is definitely provable, then it is also defeasibly provable. This is the first, simple, explanation for a defeasible provable Atom, that is covered by the previous section about definitely provable explanations.

Else, we denote the Atom and we have to a give a `Defeasible Proof`. A Defeasible Proof consists of four steps: First, we point a `Defeasible Rule` with the specified Atom as its `Head`. In the second step, we explain why the `Body` of that rule is defeasible provable (if it consists of many Atoms, then a separate proof is given for every one of them). The third step is to show that the negation of this Atom is not definitely provable (see Section 7.4). Finally, in the fourth step, we show that all rules with the negation of the Atom in their head (attacking rules) can be defeated. We call these attacking rules as `Blocked`. We characterize attacking rules as `Blocked` in two cases:

– When they cannot fire, so we must prove that their body is not defeasible provable (in case of multiple Atoms it is enough to show that only one of them is not defeasible provable). For not defeasible provable explanation, look at the Section 7.5 below.

```
<Defeasibly_provable>
    <Atom>
        <Op>  rich </Op>
        <Ind> Bob  </Ind>
    </Atom>
    <Defeasible_Proof>
        <Defeasible_rule Label="r1"> . . . </Defeasible_rule>
        <Defeasible_provable>
            <Atom>
                <Op>  wins_lotto </Op>
                <Ind> Bob        </Ind>
            </Atom>
            <Defeasible_Proof> . . . </Defeasible_Proof>
        </Defasible_provable>
        <Not_Definitely_provable>
            <Atom>
                <Not>
                    <Op>  rich </Op>
                    <Ind> Bob  </Ind>
                </Not>
            </Atom>
            <Not_Definite_Proof> . . . </Not_Definite_Proof>
        </Not_Definitely_provable>
        <Blocked>
            <Defeasible_rule Label="r3">
                <Head>
                    <Atom>
                        <Not>
                            <Op>  rich </Op>
                            <Ind> Bob  </Ind>
                        </Not>
                    </Atom>
                </Head>
                <Body> . . . </Body>
            </Defeasible_rule>
            <Superior>
                <Defeasible_rule Label="r1"/>
            </Superior>
        </Blocked>
        <Blocked>
            <Defeasible_rule Label="r4"> . . . </Defeasible_rule>
            <Not_Defeasibly_provable> . . . </Not_Defeasibly_provable>
        </Blocked>
    </Defeasible_Proof>
</Defeasible_provable>
```

Fig. 6. Example of a typical defeasible proof explanation.

– When the rule is defeated by a superior rule. Even if the body of the rule is provable (the rule fires), another rule with the specified Atom as its head fires and is declared as superior to the attacking rule. In our scheme, we just need to declare the rule that is superior to the attacking rule, and in case that this rule is different than the rule we first used as supportive, we also add the defeasible provable explanations for its body.

So, for every attacking rule we create a `Blocked` tag with the explanation of why the rule is defeated (one of the above two cases). Fig. 6 shows the structure of a definite proof explanation.

7.4. Not definitely provable explanations

The next case is the explanation of an Atom that is not definitely provable. According to our XML schema, we first denote the Atom that is not definitely provable and then we give the `NonDefinitely Proof`. The `NonDefinitely Proof` consists of all the strict rules with head equal to the negation of the nonprovable Atom, with an explanation of why they cannot fire. Inside `Blocked` tags, we include each strict rule with a NonDefinitely Provable explanation for their body. Fig. 7 demonstrates an example of a nondefinitely provable explanation.

7.5. Not defeasibly provable explanations

Finally, we describe the case when an Atom cannot be defeasibly proved. For a `NonDefeasible Proof`, firstly we have to prove that this Atom is not definitely provable (as described in the previous section). Next, we need to support our explanation with one of the following three cases:

```
<Not_Definitely_provable>
    <Atom>
        <Op>  rich </Op>
        <Ind> Bob  </Ind>
    </Atom>
    <Not_Definite_Proof>
        <Strict_rule Label="r3">
            <Head>
                <Atom>
                    <Not>
                        <Op>  rich </Op>
                        <Ind> Bob  </Ind>
                    </Not>
                </Atom>
            </Head>
            <Body>
                <Atom>
                    <Op>  owns_money </Op>
                    <Ind> Bob        </Ind>
                </Atom>
            </Body>
        </Strict_rule>
        <Not_Definitely_provable>
            <Atom>
                <Op>  owns_money </Op>
                <Ind> Bob        </Ind>
            </Atom>
            <Not_Definite_Proof> </Not_Definite_Proof>
        </Not_Definitely_provable>
    </Not_Definite_Proof>
</Not_Definitely_provable>
```

Fig. 7. Example of a nondefinitely provable explanation.

– All rules with the specified Atom as their head do not fire. For that case, we include inside `Blocked` tags every supportive defeasible rule, and also a not defeasibly provable explanation for their body.
– The negation of this Atom is definitely provable.
– We denote a rule with the negation of the specified Atom as its head, which is `Undefeated`. That means that there is no attacking rule that can defeat it. So, we embody inside `Undefeated` tags the defeasible rule that is undefeated, the defeasible provable explanation for the body of that rule and finally every attacking rule that supports the specified Atom is denoted inside `Blocked` tags either as `Not Superior` rule compared with the undefeated rule, or its body as non defeasible provable.

Fig. 8 shows an example of a nondefeasible provable explanation.

```
<Not_Defeasibly_provable>
    <Atom>
        <Op>  rich </Op>
        <Ind> Bob  </Ind>
    </Atom>
    <Not_Defeasible_Proof>
        <Not_Definitely_provable>
            <Atom>
                <Op>  rich </Op>
                <Ind> Bob  </Ind>
            </Atom>
            <Not_Definite_Proof> . . . </Not_Definite_Proof>
        </Not_Definitely_provable>
        <Undefeated>
            <Defeasible_rule Label="r4">
                <Head>
                    <Atom>
                        <Not>
                            <Op>  rich </Op>
                            <Ind> Bob  </Ind>
                        </Not>
                    </Atom>
                </Head>
                <Body> . . . </Body>
            </Defeasible_rule>
            <Defeasibly_provable> . . . </Defeasibly_provable>
            <Blocked>
                <Not_Superior>
                    <Defeasible_rule Label="r1">
                        <Head>
                            <Atom>
                                <Op>  rich </Op>
                                <Ind> Bob  </Ind>
                            </Atom>
                        </Head>
                        <Body> . . . </Body>
                    </Defeasible_rule>
                </Not_Superior>
            </Blocked>
        </Undefeated>
    </Not_Defeasible_Proof>
</Not_Defeasibly_provable>
```

Fig. 8. Example of a nondefeasible provable explanation.

8. Related work

Besides teaching logic [25], not much work has been centered around explanation in reasoning systems so far. Rule-based expert systems have been very successful in applications of AI, and from the beginning, their designers and users have noted the need for explanations in their recommendations. In expert systems like [26] and *Explainable Expert System* [27], a simple trace of the program execution/rule firing appears to provide a sufficient basis on which to build an explanation facility and they generate explanations in a language understandable to its users.

Work has also been done in explaining the reasoning in *description logics* [28,29]. This research presents a logical infrastructure for separating pieces of logical proofs and automatically generating follow-up queries based on the logical format.

8.1. Inference web

The most prominent work on proofs in the Semantic Web context is *Inference Web* [30]. The Inference Web (IW) is a Semantic Web based knowledge provenance infrastructure that supports interoperable explanations of sources, assumptions, learned information, and answers as an enabler for trust. It supports provenance, by providing proof metadata about sources, and explanation, by providing manipulation trace information. It also supports trust, by rating the sources about their trustworthiness.

The Inference Web consists of the following main components:

– *Proof Markup Language* (*PML* [31]) is an OWL-based specification for documents representing both proofs and proof meta information. Proofs are specified in PML and are interoperable.
– *IWBase* is an infrastructure within the Inference Web framework for proof meta information. It is a distributed repository of PML documents describing provenance information about proof elements such as sources, inference engines and inference rules.
– *IWExplainer* is a tool for abstracting proofs into more understandable formats.
– *IWBrowser* can display both proofs and explanations in number of proof styles and sentence formats.

IW simply requires inference rule registration and PML format. It does not limit itself to only extracting deductive engines. It provides a proof theoretic foundation on which to build and present its explanations, but any question answering system may be registered in the Inference Web and thus explained. So, in order to use the Inference Web infrastructure, a query answering system must register in the IWBase its inference engine along with its supported inference rules, using the PML specification format. The IW supports a proof generation service that facilitates the creation of PML proofs by inference engines.

It is an interesting and open issue how our implemented proof system could be registered in the Inference Web, so as to produce PML proofs. This would possibly require the registration of our inference engine, that is a Defeasible Logic reasoner, along with the corresponding inference rules, which are used in the Defeasible Logic proof theory and the explanations produced by our proof system.

Extra work needs to be done in Inference Web in order to support why-not questions. Current IW infrastructure cannot support explanations in negative answers about predicates. This is the case that corresponds to our system's explanations when an atom is not definitely or defeasibly provable.

9. Conclusion and future work

We presented a new system that aims to increase the trust of the users for the Semantic Web applications. The system automatically generates an explanation for every answer to user's queries, in a formal and useful representation. It can be used by individual users who want to get a more detailed explanation from a reasoning system in the Semantic Web, in a more human readable way. Also, an explanation could be fed into a proof checker to verify the validity of a conclusion; this is important in a multi-agent setting.

Our reasoning system is based on Defeasible Logic (a nonmonotonic rule system) and we used the related implemented meta-program and XSB as the reasoning engine. We developed a pruning algorithm that reads the XSB trace and removes the redundant information in order to formulate a sensible proof. Furthermore, the system can be used by agents, a common feature of many applications in the Semantic Web. Another contribution of our work is a RuleML extension for a formal representation of an explanation using Defeasible Logic. Additionally, we provide a web style representation for the facts, that is an optional reference to a URL. We expect that our system can be used by multiple applications, mainly in e-commerce and agent-based applications.

There are interesting ways of extending this work. The explanation can be improved to become more intuitive and human-friendly, to suit users unfamiliar with logic. Also, what-if and how-to queries could be supported. The XML Schema should be made fully compatible with the latest version of RuleML. Finally, integration with the Inference Web infrastructure will be explored.

Acknowledgement

This work was partially supported by the REWERSE Network of Excellence.

References

[1] D.L. McGuinness, F. van Harmelen, OWL Web Ontology Language Overview W3C Recommendation, 2004, http://www.w3.org/TR/owl-features/.

[2] B.N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: WWW, 2003, pp. 48–57.

[3] A.Y. Levy, M.C. Rousset, Combining Horn rules and description logics in CARIN, Artif. Intell. 104 (1–2) (1998) 165–209.

[4] R. Rosati, On the decidability and complexity of integrating ontologies and rules, WSJ 3 (1) (2005) 41–60.

[5] M. Sintek, S. Decker, TRIPLE – a query, inference, and transformation language for the Semantic Web, in: International Semantic Web Conference, 2002, pp. 364–378.

[6] I. Horrocks, P.F. Patel-Schneider, A proposal for an OWL Rules Language, in: WWW'04: Proceedings of the 13th International Conference on WorldWideWeb, ACM Press, New York, NY, USA, 2004, pp. 723–731.

[7] RuleML: The RuleML Initiative website, 2006, http://www.ruleml.org/.

[8] G. Antoniou, A. Bikakis, DR-Prolog: a system for defeasible reasoning with rules and ontologies on the Semantic Web, IEEE Trans. Know. Data Eng. 19 (2) (2007) 233–245.

[9] N. Bassiliades, N. Antonioux, I.P. Vlahavas, DR-DEVICE: a Defeasible Logic system for the Semantic Web, in: PPSWR, 2004, pp. 134–148.

[10] B.N. Grosof, M.D. Gandhe, T.W. Finin, SweetJess: translating DAMLRuleML to JESS, in: RuleML, 2002.

[11] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, dlvhex: a system for integrating multiple semantics in an answer-set programming framework, in: WLP, 2006, pp. 206–210.

[12] G. Governatori, A. Rotolo, G. Sartor, Temporalised normative positions in defeasible logic, in: ICAIL'05: Proceedings of the 10th International Conference on Artificial intelligence and Law, ACM Press, New York, NY, USA, 2005, pp. 25–34.

[13] G. Governatori, A. Rotolo, Defeasible Logic: agency, intention and obligation, in: DEON, 2004, pp. 114–128.

[14] T. Skylogiannis, G. Antoniou, N. Bassiliades, G. Governatori, DR-NEGOTIATE – a system for automated agent negotiation with Defeasible Logic-based strategies, in: EEE'05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), Washington, DC, USA, IEEE Computer Society, 2005, pp. 44–49.

[15] G. Antoniou, T. Skylogiannis, A. Bikakis, M. Doerr, N. Bassiliades, DR-BROKERING: a semantic brokering system, Know. Based Syst. 20 (1) (2007) 61–72.

[16] Daniel J. Weitzner, Jim Hendler, Tim Berners-Lee, Dan Connolly, Creating a policy-aware web: discretionary, rule-based access for the world wide web, Web and Information Security, IRM Press, 2005.

[17] G. Antoniou, D. Billington, G. Governatori, M.J. Maher, Representation results for defeasible logic, ACM Trans. Comput. Logic 2 (2) (2001) 255–287.

[18] B.N. Grosof, Prioritized conflict handing for logic programs, in: ILPS'97: Proceedings of the 1997 International Symposium on Logic Programming, MIT Press, Cambridge, MA, USA, 1997, pp. 197–211.

[19] G. Antoniou, M.J. Maher, D. Billington, Defeasible Logic versus logic programming without negation as failure, J. Log. Program. 42 (1) (2000) 47–57.

[20] M.J. Maher, A model-theoretic semantics for Defeasible Logic, in: Paraconsistent Computational Logic, 2002, pp. 67–80.

[21] G. Governatori, M.J. Maher, G. Antoniou, D. Billington, Argumentation semantics for Defeasible Logic, J. Log. Comput. 14 (5) (2004) 675–702.

[22] G. Antoniou, D. Billington, G. Governatori, M.J. Maher, Embedding defeasible logic into logic programming, Theory Pract. Log. Program. 6 (6) (2006) 703–735.

[23] M.J. Maher, A. Rock, G. Antoniou, D. Billington, T. Miller, Efficient defeasible reasoning systems, Int. J. Artif. Intell. Tool 10 (4) (2001) 483–501.

[24] JADE: Java Agent Development Framework, 2006, http://jade.tilab.com/.

[25] J. Barwise, J. Etchemendy, The language of first-order logic, Center for the Study of Language and Information, 1993.

[26] E. Shortliffe, Computer-based Medical Consultations: MYCIN, American Elsevier, New York, 1976.

[27] W. Swartout, C. Paris, J. Moore, Explanations in knowledge systems: design for explainable expert systems, IEEE Expert: Intell. Syst. Appl. 06 (3) (1991) 58–64.
[28] D.L. McGuinness, A. Borgida, Explaining subsumption in description logics, in: IJCAI (1), 1995, pp. 816–821.
[29] D. McGuinness, Explaining reasoning in Description Logics, Ph.D. Thesis, New Brunswick, NJ, 1996.
[30] D.L. McGuinness, P.P. da Silva, Explaining answers from the Semantic Web: the Inference Web approach, J. Web Sem. 1 (4) (2004) 397–413.
[31] P.P. da Silva, D.L. McGuinness, R. Fikes, A proof markup language for Semantic Web services, Inform. Syst. 31 (4) (2006) 381–395.

Grigoris Antoniou is Professor of Computer Science at the University of Crete, Greece, and Head of the Information Systems Laboratory at FO.R.T.H., the top-rated Greek Research Institute involved in many European projects. Before joining FO.R.T.H., he held professorial positions at Griffith University, Australia, and the University of Bremen, Germany.
His research interests lie in knowledge representation and reasoning, and its applications to the Semantic Web, e-commerce, digital preservation and ambient intelligence. He has published over 150 technical papers in scientific journals and conferences. He is author of three books with prestigious international publishers (Addison-Wesley and MIT Press); his book "A Semantic Web Primer" is the standard textbook in this area.He participates in a number of research projects; among them the European projects REWERSE (reasoning on the web) and CASPAR (digital preservation).In 2006, he was elected ECCAI Fellow, joining the prestigious list of the top European researchers in artificial intelligence.

Antonis Bikakis is a doctoral student at the Computer Science Department of the University of Crete, and member of the Information Systems Laboratory at the FO.R.T.H. Research Institute. He holds a M.Sc. in Computer Science from the University of Crete and a degree in Electrical and Computer Engineering from the Aristotle University of Thessaloniki.
His main interests lie in the area of Knowledge Representation and Nonmonotonic Reasoning. His current research activities concern the study of algorithms for collaborative reasoning in distributed systems, and the development of context-aware applications for ambient environments. He has published a number of scientific journal and conference papers on Defeasible Reasoning and its application to Reasoning on the Semantic Web, and on Ambient Computing Systems.

Nikos Dimaresis is a postgraduate student at the Computer Science Department of the University of Crete.

Manolis Genetzakis is a postgraduate student at the Computer Science Department of the University of Crete.

Giannis Georgalis is a postgraduate student at the Computer Science Department of the University of Crete.

Guido Governatori received his Ph.D. in Computer Science and Law at the University of Bologna in 1997. Since then he has held academic and research positions at Imperial College, Griffith University and Queensland University of Technology, The University of Queensland and NICTA. He has published over 120 scientific papers in logic, artificial intelligence, database and information systems. He was the guest editor for a few special issues on Contract Architectures and Languages. His current research interests include modal and nonclassical logics, defeasible reasoning and its application to normative reasoning and e-commerce, agent systems, business process modelling for regulatory compliance. He is a member of the editorial board of Artificial Intelligence and Law, and the leader of several Australian research projects.

Efie Karouzaki is a postgraduate student at the Computer Science Department of the University of Crete.

Nikolas Kazepis is a postgraduate student at the Computer Science Department of the University of Crete.

Dimitris Kosmadakis is a postgraduate student at the Computer Science Department of the University of Crete.

Manolis Kritsotakis is a postgraduate student at the Computer Science Department of the University of Crete.

Giannis Lilis is a postgraduate student at the Computer Science Department of the University of Crete.

Antonis Papadogiannakis is a postgraduate student at the Computer Science Department of the University of Crete.

Panagiotis Pediaditis is a postgraduate student at the Computer Science Department of the University of Crete.

Constantinos Terzakis is a postgraduate student at the Computer Science Department of the University of Crete.

Rena Theodosaki is a postgraduate student at the Computer Science Department of the University of Crete.

Dimitris Zeginis is a postgraduate student at the Computer Science Department of the University of Crete.