

Proof Explanation for the Semantic Web Using Defeasible Logic

Project Report in CS 566

Nikos Dimareisis, Antonis Papadogiannakis, Dimitris Kosmadakis,
Rena Theodosaki, Giannis Lilis, Panagiotis Pediaditis, Nikolas Kazepis,
Efie Karouzaki, Giannis Georgalis, Dimitris Zeginis,
Manolis Kritsotakis, Manolis Genetzakis, Constantinos Terzakis

October 11, 2006

1 Abstract

In this work we present the design and implementation of a new system for proof explanation in the Semantic Web, using defeasible logic. 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 convert it to a meaningful representation. It presents the explanation of an answer for a user's query back to him using a graphical interface, and also it can use an XML representation for agent communication, that is a common scenario in the Semantic Web. One of the main benefits of our system is that it supports explanations in defeasible logic for both positive and negative answers in user queries. In the remaining of this report we present the design and implementation of the system, a novel XML language for the representation of a proof explanation, and we give a variety of examples and use cases of our system.

Contents

1	Abstract	1
2	The Semantic Web Initiative	5
3	Rules on the Semantic Web	6
4	Basics of Defeasible Logics	7
4.1	Basic Characteristics	7
4.2	Syntax	7
4.3	Proof Theory	8
4.4	Reasoning Systems	11
5	The Proof Layer	11
6	Explanation Use Cases	12
6.1	Use Case: Online Shop and E-commerce	12
6.2	Use Case: University Graduate	13
7	Extension of RuleML for Explanation Representation	13
7.1	Atoms, Facts and Rule Representation	13
7.2	Definitely Provable Explanations	14
7.3	Defeasibly Provable Explanations	15
7.4	Not Definitely Provable Explanations	17
7.5	Not Defeasibly Provable Explanations	17
8	Some Concrete Examples	19
8.1	Example1: Lunch Time	19
8.2	Example3: Marine Biology	20
8.3	Example4: Criminality	25
9	Proof tree construction	27
10	Proof Tree Pruning	28
10.1	Examples of the proof tree pruning	30
11	Graphical user interface to the proof system	34
12	Agent interface to the proof system	35
12.1	Architecture	35
12.2	Visual Agent	38
12.3	Command Line Agent	43
12.3.1	Operation of the CLAgent	43
12.4	Agent Use Cases	44

13	Relative Work	46
13.1	Inference Web	46
14	Conclusions and Future Work	48
A	XML Schema for Explanation Representation	50

List of Figures

1	Layers of the Semantic Web	6
2	Declaration of an Atom	14
3	Declaration of a Defeasible Rule	14
4	Example of a typical Definite Proof Explanation	15
5	Example of a typical Defeasible Proof Explanation	16
6	Example of a Non Definitely Provable Explanation	17
7	Example of a Non Defeasible Provable Explanation	18
8	The marked rule is pruned	30
9	The marked rule is pruned	31
10	The colored segments correspond to the respective pruned and unpruned versions of the tree	31
11	The colored segments correspond to the respective pruned and unpruned versions of the tree	32
12	The colored segments correspond to the respective pruned and unpruned versions of the tree. In the unpruned tree, we show all rules that actually fail due to the unblocked attacking rule, whereas in the pruned tree, we just show that rule and the fact that it cannot be defeated	32
13	The list pruning can be seen in the colored segments, corresponding respectively to the pruned and unpruned versions. The unpruned tree shows the recursive nature of the Prolog list (a list in prolog is the head element and the remaining sublist), whereas in the pruned tree we simply present the list elements	33
14	The handling of the missing proof can be seen in the colored segments, corresponding respectively to the pruned and unpruned versions. We simply clone the specific proof (red) where needed (green)	33
15	The system architecture	35
16	Error message if Responder Agent is not entered	38
17	Error message if the question is not entered	39
18	The answer to a question	40
19	The proof of a question	41
20	Error messages occur while computing the result	42
21	Sample configuration file	43
22	Communication of the main Agent and the CLAgent	43
23	Main Agent sends the answer or proof	44
24	Log file of the CLAgent	45

2 The Semantic Web Initiative

The aim of the Semantic Web initiative [1] is to advance the state of the current Web through the use of semantics. More specifically, it proposes to use semantic annotations to describe the meaning of certain parts of Web information. For example, the Web site of a hotel could be suitably annotated to distinguish between hotel name, location, category, number of rooms, available services etc. Such metadata could facilitate the automated processing of the information on the Web site, thus making it accessible to machines and not primarily to human users, as it is the case today.

The development of the Semantic Web proceeds in steps, each step building a layer on top of another. The layered design is shown in Figure 1, which is outlined below.

- At the bottom layer we find XML [2], a language that lets one write structured Web documents with a user-defined vocabulary. XML is particularly suitable for sending documents across the Web, thus supporting syntactic interoperability.
- RDF is a basic data model for writing simple statements about Web objects (resources). The RDF data model does not rely on XML, but RDF has an XML-based syntax. Therefore it is located on top of the XML layer.
- RDF Schema provides modeling primitives for organizing Web objects into hierarchies. RDF Schema is based on RDF. RDF Schema can be viewed as a primitive language for writing ontologies.
- But there is a need for more powerful ontology languages that expand RDF Schema and allow the representation of more complex relationships between Web objects. Ontology languages, such as OWL, are built on the top of RDF and RDF Schema.
- The logic layer is used to enhance the ontology language further, and to allow writing application-specific declarative knowledge.
- The proof layer involves the actual deductive process, as well as the representation of proofs in Web languages and proof validation. o Finally trust will emerge through the use of digital signatures, and other kind of knowledge, based on recommendations
- Finally trust will emerge by using digital signatures, and other kind of knowledge, based on recommendations by agents we trust, or rating and certification agencies and consumer bodies.

For an easy yet comprehensive introduction to the Semantic Web see [3].

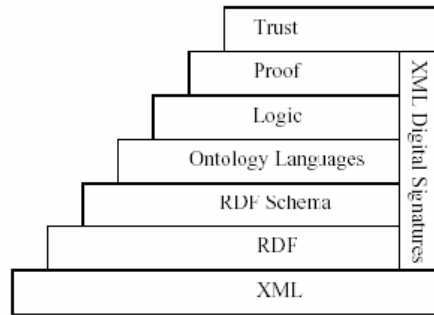


Figure 1: Layers of the Semantic Web

3 Rules on the Semantic Web

At present, the highest layer that has reached sufficient maturity is the ontology layer in the form of the description logic-based language OWL [4]. 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, such as knowledge bases, etc.

Apart from the classical rules that lead to monotonic logical systems, recently researchers started to study systems capable of handling conflicts among rules. Generally speaking, the main sources of such conflicts are:

- Default inheritance within ontologies.
- Ontology merging, where knowledge from different sources is combined.
- Rules with exceptions as a natural representation of business rules.
- Reasoning with incomplete information.

4 Basics of Defeasible Logics

4.1 Basic Characteristics

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 [5] and Courteous Logic Programs [6]. 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 [7]).
- 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.

4.2 Syntax

A *defeasible theory* D 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 $A \rightarrow p$ 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:

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

Inference from facts and strict rules only is called *definite inference*. 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

$$faculty(X) \Rightarrow tenured(X)$$

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

$$heavy(X) \rightsquigarrow \neg flies(X)$$

which reads as follows: “If an animal is heavy then it may not be able to fly”. The main point is that the information that an animal is heavy is not sufficient evidence to conclude that it doesn’t fly. It is only evidence that the animal *may* not be able to fly.

A superiority relation on R is an acyclic relation $>$ on R (that is, the transitive closure of $>$ is irreflexive). When $r1 > r2$, then $r1$ is called *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 one 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.

4.3 Proof Theory

A *conclusion* of a defeasible theory D is a tagged literal. Conventionally there are four tags, so a conclusion has 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 .

Provability is based on the concept of a *derivation* (or *proof*) in $D = (F, R, >)$. A derivation is a finite sequence $P = (P(1), \dots, P(n))$ 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. $(P(1..i))$ 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
 $\exists r \in R_s[q] \forall a \in A(r): +\Delta a \in P(1..i)$

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).

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

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 ($-\Delta a$).

Defeasible provability requires consideration of chains of reasoning for the complementary literal, and possible resolution using the superiority relation. Thus the inference rules for defeasible provability are more complicated than those for definite provability.

$+\partial$: We may append $P(i+1)=+\partial q$ if either
(1) $+\Delta q \in P(1..i)$ or
(2) (2.1) $\exists r \in R_{sd}[q] \forall a \in A(r): +\partial a \in P(1..i)$ and
(2.2) $-\Delta \sim q \in P(1..i)$ and
(2.3) $\forall s \in R[\sim q]$ either
(2.3.1) $\exists a \in A(s): -\partial a \in P(1..i)$ or
(2.3.2) $\exists t \in R_{sd}[q]$ such that
 $\forall a \in A(t): +\partial a \in P(1..i)$ 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 inference rule $-\partial$. It is a strong negation of the inference rule $+\partial$ [2].

- $-\partial$: We may append $P(i+1) = -\partial q$ if
- (1) $-\Delta q \in P(1..i)$ and
 - (2) (2.1) $\forall r \in R_{sd}[q] \exists a \in A(r): -\partial a \in P(1..i)$ or
 - (2.2) $+\Delta \sim q \in P(1..i)$ or
 - (2.3) $\exists s \in R[\sim q]$ such that
 - (2.3.1) $\forall a \in A(s): +\partial a \in P(1..i)$ and
 - (2.3.2) $\forall t \in R_{sd}[q]$ either
 - $\exists a \in A(t): -\partial a \in P(1..i)$ or $t \not\prec s$

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).

A more detailed definition of the proof theory is found in [5]. A model theoretic semantics is found in [8], and argumentation semantics is discussed in [4].

4.4 Reasoning Systems

Recent system implementations, capable of reasoning with monotonic rules, non-monotonic rules, priorities, RDF [9] data and RDF Schema [10] ontologies, are DR-Prolog [11] and DR-DEVICE [12]. DR-Prolog is a defeasible reasoning system for reasoning on the Web. Its main characteristics are the following:

- Its user interface is compatible with RuleML, the main standardization effort for rules on the Semantic Web.
- It is based on Prolog. The core of the system consists of a well-studied translation of defeasible knowledge into logic programs under Well-Founded Semantics.
- The main focus is on flexibility. Strict and defeasible rules and priorities are part of the interface and the implementation.
- The system can reason with rules and ontological knowledge written in RDF Schema (RDFS) or OWL.

DR-DEVICE is also a defeasible reasoning system for reasoning on the Web. Its main characteristics are the following:

- Its user interface is compatible with RuleML
- It is based on a CLIPS-based implementation of deductive rules. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes.

5 The Proof Layer

The next subjects in the development of the Semantic Web, apart from becoming a global database, are the issues of accessibility, trust and credibility. These upper levels have not been researched enough and these are the next challenges for the Semantic Web. The next step in the architecture of the Semantic Web is the proof layer. Little has been written and done for this layer, but it is expected to become very important in the near future.

Because not all the data sources will be considered equally reliable, when the users (humans and computer agents) receive an answer from a Semantic Web application, they need to know the data sources that were used and the sources' reliability. Thus it is necessary for the users to be able to evaluate an answer for its trustworthiness. The solution is to provide *explanations* for the derivation history, which is the series of inference steps that can be retraced. An explanation will trace an answer back to a given set of facts and the inference rules used.

Users, in order to trust an answer which receive from an application, they may need to inspect the whole deductive proof trace that was used to derive implicit

information. Usually proof traces are too long and complex and more suitable in understanding for expert logicians. So it is better to reduce the explanation from the proof traces to a more understandable form for the users. Furthermore, because all the facts that are contained in a proof could be assigned to a source, users need to retrieve some information about the sources in order to trust them. Perhaps applications that rate sources about their authoritativeness could be developed to help for this reason. Finally, applications need to be developed that provide to users a user interface that give them the ability to ask questions and request explanations in answers and present them the proofs and the corresponding explanations.

In this project we implement a defeasible reasoning system for reasoning on the Web, which provides the additional capability of presenting explanations to users for the answers to their queries.

6 Explanation Use Cases

In this section, we mention two examples where the explanation in the Semantic Web is used.

6.1 Use Case: Online Shop and E-commerce

Suppose an agent, which represents an online shop, sends to Bob's agent a message that he owns 30 Euros. Last week he purchased a DVD from the online shop, that costs 30 Euros. The item was delivered to his address and Bob's agent was notified that Bob must pay the item's price. Then Bob ask his agent the reason why he owes that cost and the shop's agent respond with an explanation, which is the following sequence of facts and rules:

- $purchase(Customer, Item), price(Item, Cost), delivered(Item, Customer) \rightarrow owes(Customer, Price)$
This is a rule from the shops terms and conditions
- $purchase(Bob, DVD)$ The Web log of Bob's purchase
- $price(Item, Cost)$ The price of the DVD in the online shop
- $delivered(Item, Customer)$ Delivery proof of DVD to Bob with a courier number

6.2 Use Case: University Graduate

Suppose Jim is a student in a university. His agent asks the agent of the university's secretary, if Jim is able to graduate from the department that he attends. The secretary gives him a negative answer and the student requests his agent to receive an explanation about the reason he cannot graduate. The secretary's agent gives as an explanation the following sequence of facts and rules:

- $completed_units(Student), passed_optional_lessons(Student), undergraduate_thesis(Student) \rightarrow graduate(Student)$
This is a rule from the department's program studies
- $\neg passed_optional_lessons(Jim)$ The list with the lessons that Jim has passed, where is referred that he has not passed the required number of optional lessons in order to graduate

Then Jim makes a follow up question to retrieve a further explanation about the number of optional lessons that he has passed and the required number in order to graduate.

- $passed(Student, X), optional(X), passed(Student, Y), optional(Y), passed(Student, Z), optional(Z), notSame(X, Y), notSame(Y, Z), notSame(X, Z) \rightarrow passed_optional_lessons(Student)$
This is a rule from the department's program studies. A student must pass at least three different optional lessons. We prefer to use the predicate *notSame* instead of expressing inequality in logical terms.
- $passed(Jim, CS110) optional(CS110)$ Jim has passed the optional lesson with the code CS110
- $passed(Jim, CS231) optional(CS231)$ Jim has passed the optional lesson with the code CS231. So Jim has only passed two optional lessons.

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. The complete schema specification is given in Appendix A.

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). Figure 2 shows the declaration of a typical Atom.

```

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

```

Figure 2: Declaration of an Atom

Similarly to RuleML, a *Fact* is consisted by an Atom that comprise a certain knowledge. The last primitive entity of our schema is *Rules*. In defeasible logic, we distinguish two kinds of Rules: *Strict Rules* and *Defeasible Rules*. In our schema we also note with a different element these two kind of rules. Both kinds consists 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. Figure 3 shows a typical example of a Defeasible Rule.

```

<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>

```

Figure 3: Declaration of a Defeasible Rule

7.2 Definitely Provable Explanations

The simplest proof explanation is in 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 give a `Strict Rule` with `Head` this Atom and `Body` an Atom that should be also proved definitely with the same way. If the `Body` consists of multiple Atoms, then we state the definite provable explanation for every atom of the `Body`. Figure 4 shows the structure of a definite proof explanation.

```

<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>

```

Figure 4: 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 give a Defeasible Proof. A Defeasible Proof consists of four steps: First, we point a Defeasible Rule with Head the specified Atom. 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 have to show that all the rules with head the negation of the Atom that we prove (attacks) can be defeated. We call these attack rules as Blocked. We characterize an attack rule 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.

- When the rule is defeated by a superiority relation. Even if the body of the rule is provable (the rule fires), an other rule with head the Atom that we prove and fires should be declared as superior to the attack rule. In our scheme, we just need to declare the rule that is superior to the attack 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 attack rule we create a Blocked tag with the explanation of why the rule is defeated (one of the above two cases). Figure 5 shows the structure of a definite proof explanation.

```

<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>
    </Defeasible_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>

```

Figure 5: Example of a typical Defeasible 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 `Non Definitely Proof`. The `Non Definitely Proof` consists of all the strict rules with head equal to the negation of the non provable Atom, with an explanation of why they cannot fire. Inside `Blocked` tags, we include each strict rule with a `Non Definitely Provable` explanation for their body. Figure 6 demonstrates an example of a non definitely provable explanation.

```
<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>
```

Figure 6: Example of a Non Definitely Provable Explanation

7.5 Not Defeasibly Provable Explanations

At last, we describe the case when an Atom cannot be defeasibly proved. For a `Non Defeasible 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:

- All the rules with head the specified Atom does not fire. For that case, we include inside `Blocked` tags every defeasible rule with head this Atom and also a not defeasibly provable explanation for their body.

- The negation of this Atom is definitely provable.
- We denote a rule with head the negation of the specified Atom that is `Undefeated`. That means that there is no attack 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 attack rule (with head the Atom that we prove, that is not defeasible provable) is denoted inside `Blocked` tags either as `Not Superior` rule compared with the undefeated rule, or its body as non defeasible provable.

Figure 7 shows an example of a non defeasible 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>

```

Figure 7: Example of a Non Defeasible Provable Explanation

8 Some Concrete Examples

In this section, we present some examples of explanation representation in our XML schema using defeasible logic reasoning.

8.1 Example1: Lunch Time

First, we demonstrate a simple example using the following rules:

```
fact(hungry(isidoros)).
fact(empty_kitchen(isidoros)).
defeasible(r1,order_pizza(X),[hungry(X),empty_kitchen(X)]).
```

The answer to the question *defeasibly(order_pizza(isidoros))* is TRUE. The explanation to that answer, according to our XML scheme, is given below:

```
1 <Defeasibly_provable>
2   <Atom>
3     <Op> order_pizza </Op>
4     <Ind> isidoros </Ind>
5   </Atom>
6   <Defeasible_Proof>
7     <Defeasible_rule Label="r1">
8       <Head>
9         <Atom>
10          <Op> order_pizza </Op>
11          <Ind> isidoros </Ind>
12        </Atom>
13      </Head>
14      <Body>
15        <Atom>
16          <Op> hungry </Op>
17          <Ind> isidoros </Ind>
18        </Atom>
19
20        <Atom>
21          <Op> empty_kitchen </Op>
22          <Ind> isidoros </Ind>
23        </Atom>
24      </Body>
25    </Defeasible_rule>
26
27    <Defeasibly_provable>
28      <Definitely_provable>
29        <Atom>
30          <Op> hungry </Op>
31          <Ind> isidoros </Ind>
32        </Atom>
33      <Definite_Proof>
34        <Fact>
35          <Atom>
36            <Op> hungry </Op>
37            <Ind> isidoros </Ind>
38          </Atom>
39        </Fact>
40      </Definite_Proof>
41    </Definitely_provable>
42  </Defeasibly_provable>
```

```

43
44     <Defeasibly_provable>
45         <Definitely_provable>
46             <Atom>
47                 <Op> empty_kitchen </Op>
48                 <Ind> isidoros </Ind>
49             </Atom>
50         <Definite_Proof>
51             <Fact>
52                 <Atom>
53                     <Op> empty_kitchen </Op>
54                     <Ind> isidoros </Ind>
55                 </Atom>
56             </Fact>
57         </Definite_Proof>
58     </Definitely_provable>
59 </Defeasibly_provable>
60
61     <Not_Definitely_provable>
62         <Atom>
63             <Not>
64                 <Op> order_pizza </Op>
65                 <Ind> isidoros </Ind>
66             </Not>
67         </Atom>
68     <Not_Definite_Proof>
69 </Not_Definite_Proof>
70 </Not_Definitely_provable>
71
72 </Defeasible_Proof>
73 </Defeasibly_provable>

```

8.2 Example3: Marine Biology

In this example, we have the following rule set:

```

strict(r1,cephalopod(X),[nautilus(X)]).
strict(r2,mollusc(X),[cephalopod(X)]).
defeasible(r3,shell(X),[nautilus(X)]).
defeasible(r4,~(shell(X)),[cephalopod(X)]).
defeasible(r5,shell(X),[mollusc(X)]).
fact(nautilus(nancy)).
sup(r5,r4)

```

If we ask *defeasibly(shell(nancy))*, the answer is TRUE. The literal *shell(nancy)* is defeasibly provable. We present below the explanation according to our scheme, that our system generates for this example.

```

1 <Defeasibly_provable>
2   <Atom>
3     <Op>shell</Op>
4     <Ind>nancy</Ind>
5   </Atom>
6 <Defeasible_Proof>
7   <Defeasible_rule Label="r3">
8     <Head>
9       <Atom>

```

```

10         <Op>shell</Op>
11         <Ind>nancy</Ind>
12     </Atom>
13 </Head>
14 <Body>
15     <Atom>
16         <Op>nautilus</Op>
17         <Ind>nancy</Ind>
18     </Atom>
19 </Body>
20 </Defeasible_rule>
21 <Defeasibly_provable>
22     <Definitely_provable>
23         <Atom>
24             <Op>nautilus</Op>
25             <Ind>nancy</Ind>
26         </Atom>
27         <Definite_Proof>
28             <Fact>
29                 <Atom>
30                     <Op>nautilus</Op>
31                     <Ind>nancy</Ind>
32                 </Atom>
33             </Fact>
34         </Definite_Proof>
35     </Definitely_provable>
36 </Defeasibly_provable>
37 <Not_Definitely_provable>
38 <Atom>
39 <Not>
40     <Op>shell</Op>
41     <Ind>nancy</Ind>
42 </Not>
43 </Atom>
44 <Not_Definite_Proof></Not_Definite_Proof>
45 </Not_Definitely_provable>
46 <Blocked>
47     <Defeasible_rule Label="r4">
48         <Head>
49             <Atom>
50                 <Not>
51                     <Op>shell</Op>
52                     <Ind>nancy</Ind>
53                 </Not>
54             </Atom>
55         </Head>
56         <Body>
57             <Atom>
58                 <Op>cephalopod</Op>
59                 <Ind>nancy</Ind>
60             </Atom>
61         </Body>
62     </Defeasible_rule>
63 <Superior>
64     <Defeasible_rule Label="r5">
65         <Head>
66             <Atom>
67                 <Op>shell</Op>
68                 <Ind>nancy</Ind>
69             </Atom>
70         </Head>
71         <Body>

```

```

72         <Atom>
73             <Op>mollusc</Op>
74             <Ind>nancy</Ind>
75         </Atom>
76     </Body>
77 </Defeasible_rule>
78 <Defeasibly_provable>
79     <Definitely_provable>
80         <Atom>
81             <Op>mollusc</Op>
82             <Ind>nancy</Ind>
83         </Atom>
84     <Definite_Proof>
85         <Strict_rule Label="r2">
86             <Head>
87                 <Atom>
88                     <Op>mollusc</Op>
89                     <Ind>nancy</Ind>
90                 </Atom>
91             </Head>
92             <Body>
93                 <Atom>
94                     <Op>cephalopod</Op>
95                     <Ind>nancy</Ind>
96                 </Atom>
97             </Body>
98         </Strict_rule>
99     <Definitely_provable>
100         <Atom>
101             <Op>cephalopod</Op>
102             <Ind>nancy</Ind>
103         </Atom>
104     <Definite_Proof>
105         <Strict_rule Label="r1">
106             <Head>
107                 <Atom>
108                     <Op>cephalopod</Op>
109                     <Ind>nancy</Ind>
110                 </Atom>
111             </Head>
112             <Body>
113                 <Atom>
114                     <Op>nautilus</Op>
115                     <Ind>nancy</Ind>
116                 </Atom>
117             </Body>
118         </Strict_rule>
119     <Definitely_provable>
120         <Atom>
121             <Op>nautilus</Op>
122             <Ind>nancy</Ind>
123         </Atom>
124     <Definite_Proof>
125         <Fact>
126             <Atom>
127                 <Op>nautilus</Op>
128                 <Ind>nancy</Ind>
129             </Atom>
130         </Fact>
131     </Definite_Proof>
132 </Definitely_provable>
133 </Definite_Proof>

```

```

134         </Definitely_provable>
135         </Definite_Proof>
136     </Definitely_provable>
137 </Defeasibly_provable>
138 </Superior>
139 </Blocked>
140 </Defeasible_Proof>
141 </Defeasibly_provable>

```

Since *defeasibly(shell(nancy))* is TRUE, the answer to the question *not_defeasibly(shell(nancy))* will be FALSE, because the literal *(shell(nancy))* is not defeasibly provable. The explanation for this negative answer is given below:

```

1 <Not_Defeasibly_provable>
2   <Atom>
3     <Not>
4       <Op>shell</Op>
5       <Ind>nancy</Ind>
6     </Not>
7   </Atom>
8 <Not_Defeasibly_Proof>
9   <Not_Definitely_provable>
10     <Atom>
11       <Not>
12         <Op>shell</Op>
13         <Ind>nancy</Ind>
14       </Not>
15     </Atom>
16     <Not_Definite_Proof></Not_Definite_Proof>
17   </Not_Definitely_provable>
18 <Blocked>
19   <Defeasible_rule Label="r4">
20     <Head>
21       <Atom>
22         <Not>
23           <Op>shell</Op>
24           <Ind>nancy</Ind>
25         </Not>
26       </Atom>
27     </Head>
28     <Body>
29       <Atom>
30         <Op>cephalopod</Op>
31         <Ind>nancy</Ind>
32       </Atom>
33     </Body>
34   </Defeasible_rule>
35 <Superior>
36   <Defeasible_rule Label="r5">
37     <Head>
38       <Atom>
39         <Op>shell</Op>
40         <Ind>nancy</Ind>
41       </Atom>
42     </Head>
43     <Body>
44       <Atom>
45         <Op>mollusc</Op>
46         <Ind>nancy</Ind>
47       </Atom>
48     </Body>

```

```

49     </Defeasible_rule>
50     <Defeasibly_provable>
51         <Definitely_provable>
52             <Atom>
53                 <Op>mollusc</Op>
54                 <Ind>nancy</Ind>
55             </Atom>
56             <Definite_Proof>
57                 <Strict_rule Label="r2">
58                     <Head>
59                         <Atom>
60                             <Op>mollusc</Op>
61                             <Ind>nancy</Ind>
62                         </Atom>
63                     </Head>
64                     <Body>
65                         <Atom>
66                             <Op>cephalopod</Op>
67                             <Ind>nancy</Ind>
68                         </Atom>
69                     </Body>
70                 </Strict_rule>
71             <Definitely_provable>
72                 <Atom>
73                     <Op>cephalopod</Op>
74                     <Ind>nancy</Ind>
75                 </Atom>
76                 <Definite_Proof>
77                     <Strict_rule Label="r1">
78                         <Head>
79                             <Atom>
80                                 <Op>cephalopod</Op>
81                                 <Ind>nancy</Ind>
82                             </Atom>
83                         </Head>
84                         <Body>
85                             <Atom>
86                                 <Op>nautilus</Op>
87                                 <Ind>nancy</Ind>
88                             </Atom>
89                         </Body>
90                     </Strict_rule>
91                 <Definitely_provable>
92                     <Atom>
93                         <Op>nautilus</Op>
94                         <Ind>nancy</Ind>
95                     </Atom>
96                 <Definite_Proof>
97                     <Fact>
98                         <Atom>
99                             <Op>nautilus</Op>
100                             <Ind>nancy</Ind>
101                         </Atom>
102                     </Fact>
103                 </Definite_Proof>
104             </Definitely_provable>
105         </Definite_Proof>
106     </Definitely_provable>
107 </Definite_Proof>
108 </Definitely_provable>
109 </Defeasibly_provable>
110 </Superior>

```



```

111     </Blocked>
112     </Not_Defeasibly_Proof>
113 </Not_Defeasibly_provable>

```

8.3 Example4: Criminality

Using the following rule set, we answer the question *defeasibly((hasGun(a))*):

```

defeasible(r3,~(hasGun(X)],[pacifist(X)]).
defeasible(r4,hasGun(X) ,[livesInChicago(X)]).
defeasible(r2,~(pacifist(X)],[republican(X)]).
defeasible(r1,pacifist(X) ,[quaker(X)]).
fact(quaker(a)).
fact(republican(a)).
fact(livesInChicago(a)).
sup(r3,r4).

```

The answer is FALSE, because the literal (hasGun(a)) cannot defeasibly proved.

```

1 <Not_Defeasibly>
2   <Atom>
3     <Not>
4       <Op>hasGun</Op>
5       <Ind>a</Ind>
6     </Not>
7   </Atom>
8   <Not_Defeasibly_Proof>
9     <Not_Definitely_provable>
10    <Atom>
11      <Not>
12        <Op>hasGun</Op>
13        <Ind>a</Ind>
14      </Not>
15    </Atom>
16    <Not_Definite_Proof></Not_Definite_Proof>
17  </Not_Definitely_provable>
18  <Blocked>
19    <Defeasible_rule Label="r3">
20      <Head>
21        <Atom>
22          <Not>
23            <Op>hasGun</Op>
24            <Ind>a</Ind>
25          </Not>
26        </Atom>
27      </Head>
28      <Body>
29        <Atom>
30          <Op>pacifist</Op>
31          <Ind>a</Ind>
32        </Atom>
33      </Body>
34    </Defeasible_rule>
35    <Not_Defeasibly_provable>
36      <Atom>
37        <Op>pacifist</Op>
38        <Ind>a</Ind>
39      </Atom>

```

```

40     <Not_Defeasibly_Proof>
41         <Not_Definitely_provable>
42             <Atom>
43                 <Op>pacifist</Op>
44                 <Ind>a</Ind>
45             </Atom>
46             <Not_Definite_Proof></Not_Definite_Proof>
47         </Not_Definitely_provable>
48     <Undefeated>
49         <Defeasible_rule Label="r2">
50             <Head>
51                 <Atom>
52                     <Not>
53                         <Op>pacifist</Op>
54                         <Ind>a</Ind>
55                     </Not>
56                 </Atom>
57             </Head>
58             <Body>
59                 <Atom>
60                     <Op>republican</Op>
61                     <Ind>a</Ind>
62                 </Atom>
63             </Body>
64         </Defeasible_rule>
65     <Defeasibly_provable>
66         <Definitely_provable>
67             <Atom>
68                 <Op>republican</Op>
69                 <Ind>a</Ind>
70             </Atom>
71         <Definite_Proof>
72             <Fact>
73                 <Atom>
74                     <Op>republican</Op>
75                     <Ind>a</Ind>
76                 </Atom>
77             </Fact>
78         </Definite_Proof>
79     </Definitely_provable>
80 </Defeasibly_provable>
81 <Blocked>
82     <Not_Superior>
83         <Defeasible_rule Label="r1">
84             <Head>
85                 <Atom>
86                     <Op> pacifist </Op>
87                     <Ind> a </Ind>
88                 </Atom>
89             </Head>
90             <Body>
91                 <Atom>
92                     <Op> quaker </Op>
93                     <Ind> nikos </Ind>
94                 </Atom>
95             </Body>
96         </Defeasible_rule>
97     </Not_Superior>
98 </Blocked>
99     </Undefeated>
100 </Not_Defeasibly_Proof>
101 </Not_Defeasibly_provable>

```

```
102         </Blocked>
103     </Not_Defeasibly_Proof>
104 </Not_Defeasibly_provable>
```

9 Proof tree construction

The foundation of the proof system lies in the prolog metaprogram that implements the rules and reflects the traits of defeasible logic, and the trace facility of the XSB implementation of prolog which is used for extracting information about the runtime behavior of the metaprogram. Thus, the trace produced by the invocation of the prolog metaprogram with the defeasible logic program as input, is used for constructing a proof tree which is subsequently used by the system in order to formulate a sensible proof.

The method chosen for communicating the runtime trace information of the metaprogram from the XSB execution environment to the Java front-end, on which the proof system along with its graphical and agent-based interfaces was implemented, was the invocation of the XSB executable from inside the Java code. Through the use of Javas *exec* method it was possible on one hand to send commands to the XSB interpreter that was running as a stand-alone process and on the other hand to receive the output that was produced as an effect.

Thus, at the initialization step, the XSB process invoker executes the XSB application through the *exec* method of the Javas *Runtime* singleton and enables the trace facility by sending the following commands to the running process:

```
trace.                /* enable trace mode */
debug_ctl(prompt, off). /* turn off trace prompt */
```

Then, in order to load the defeasible logic metaprogram to the XSB interpreter, provided that the metaprogram is contained in the *amb_metaprogram.P* file, the following command is sent:

```
[amb_metaprogram.P].
```

After the successful loading of the metaprogram, the system is ready to accept the defeasible logic based program which constitutes the metaprograms database. This is achieved by executing the XSB load file command once again with the programs filename as a parameter:

```
[defeasible_logic_program.P].
```

Subsequently, 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 that is provided by the trace. Namely:

- A string representation of the predicates name
- The predicates 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).

10 Proof Tree Pruning

The pruning algorithm utilized to produce the final tree from the initial XSB trace focuses on two major points. Firstly, the XSB trace produces a tree with redundant information that needs to be removed from the final tree. One reason for this is that we use a particular metaprogram in order to simulate the Defeasible Logic over Prolog. For the simulation to be successful, we need some additional rules which add unwanted information to the XSB trace. Another reason for the redundant information is the way prolog evaluates the rules 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 previously mentioned XML schema. In the remainder of the document we will take a closer look to the details of these two issues.

In order for the metaprogram to be able to represent the negation of the predicates and evaluate the double negation of a predicate to the predicate itself, we needed to add an extra rule, *'negation'*:

```
negation(~(X),X):-!.
negation(X,~(X)).
```

This rule, of course, provides no necessary information for the proof but solves a simple technicality. Therefore, it is cutoff by the pruning process. Furthermore, another extra rule added in the metaprogram was *'xsb_meta_not'*:

```
xsb_meta_not(X) :- not(X).
```

In Prolog, the **not** predicate does not maintain the negation property (i.e. in the trace of **not(a)** we get information about the state of *a* but not for its inverse - which is what we actually want). To overcome this problem, we added this extra rule using it as semantic information while constructing and parsing the XSB trace and building the proof tree.

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 them. For example, the rule ‘*supportive_rule*’:

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

evaluates to ‘*strict*’ or ‘*defeasible*’, meaning that a supportive rule can be either a strict rule or a defeasible rule. If ‘*supportive_rule*’ evaluates to false, then we would want to show that both ‘*strict*’ and ‘*defeasible*’ evaluated to false. On the other hand, if it evaluates to true, then we only want to keep the one that evaluated to true (or the first of the two in case of both evaluating to true).

As earlier mentioned, the tree produced by the XSB trace is not in the desired form. The techniques already shown do not suffice for this purpose. The unavoidable complexity of the metaprogram produces complicated and unintuitive proof trees. Our goal is to bring this tree to a form compatible to that dictated by the XML schema. In order to do this, we traverse the tree recursively in a depth-first manner applying the following heuristic rules to produce the final tree.

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 rule, or there is at least one strict rule having the atom as its head and is definitely provable, so we locate the first such rule along with the definite proof of its body. [Figure 8]
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’ rule as well as all the aforementioned strict rules along with the proof that their bodies are not definitely provable. [Figure 9]
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. [Figures 10, 11]

4. **Pruning Not Defeasibly** When the atom of a query cannot be proved defeasibly, it cant 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. In any case, we locate the necessary proof sub trees. [Figure 12]
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 proof tree. To remedy this case we flatten the lists to a single depth sequence of atoms. [Figure 13]
6. **Handling missing proofs** XSB uses a caching technique in order to avoid reevaluating already evaluated expressions. Effectively, this means that the first time we encounter 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 doesnt show the whole execution tree. In some cases, the aforementioned 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 unacceptable, so we are forced to keep a copy of the initial trace so as to recover a possibly pruned predicate evaluation subtree. [Figure 14]

Using these heuristic techniques, we end up with a version of the proof 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. However, the drawback is that these heuristics are fully dependent on the metaprogram. Any changes in the metaprogram would necessitate changes in the pruning implementation. It would be interesting to consider the possibility of automating the process of implementing the heuristic pruning, based on the metaprogram, but this exceeds the aims of the current study.

10.1 Examples of the proof tree pruning

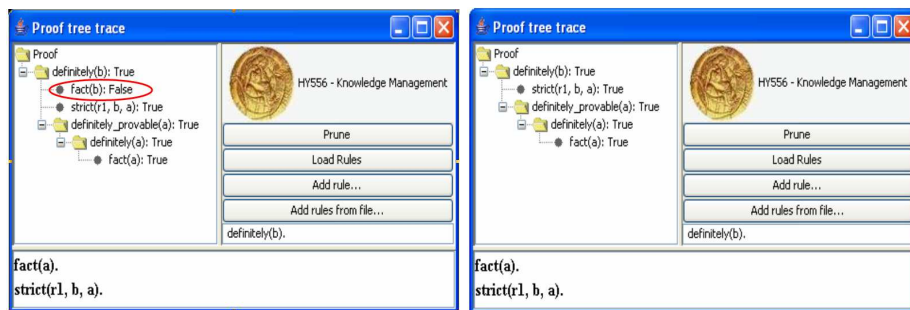


Figure 8: The marked rule is pruned

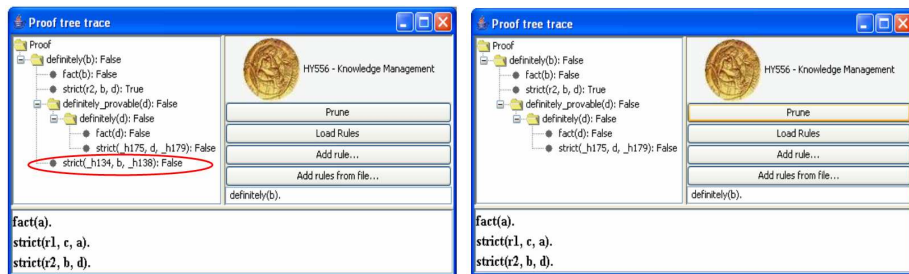


Figure 9: The marked rule is pruned

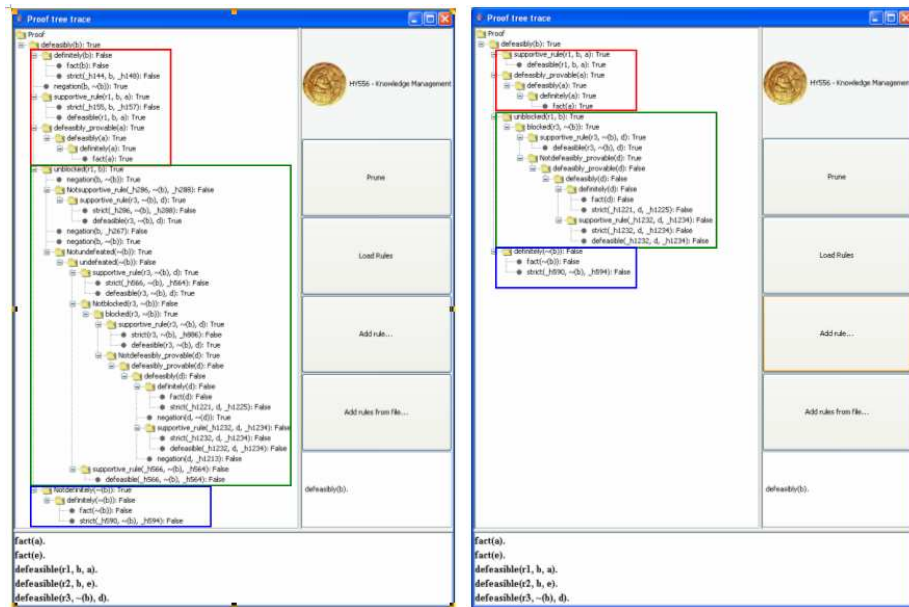


Figure 10: The colored segments correspond to the respective pruned and unpruned versions of the tree

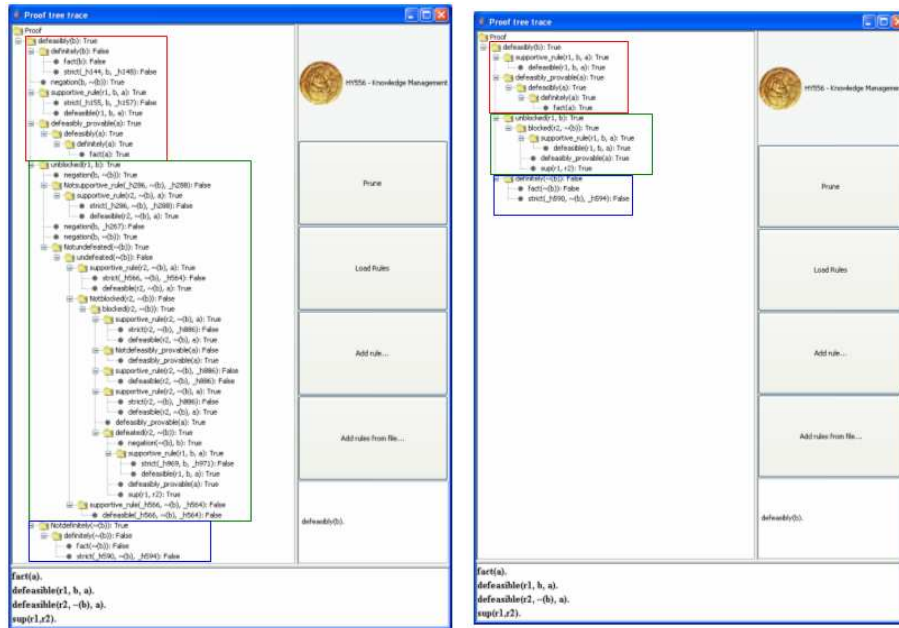


Figure 11: The colored segments correspond to the respective pruned and unpruned versions of the tree

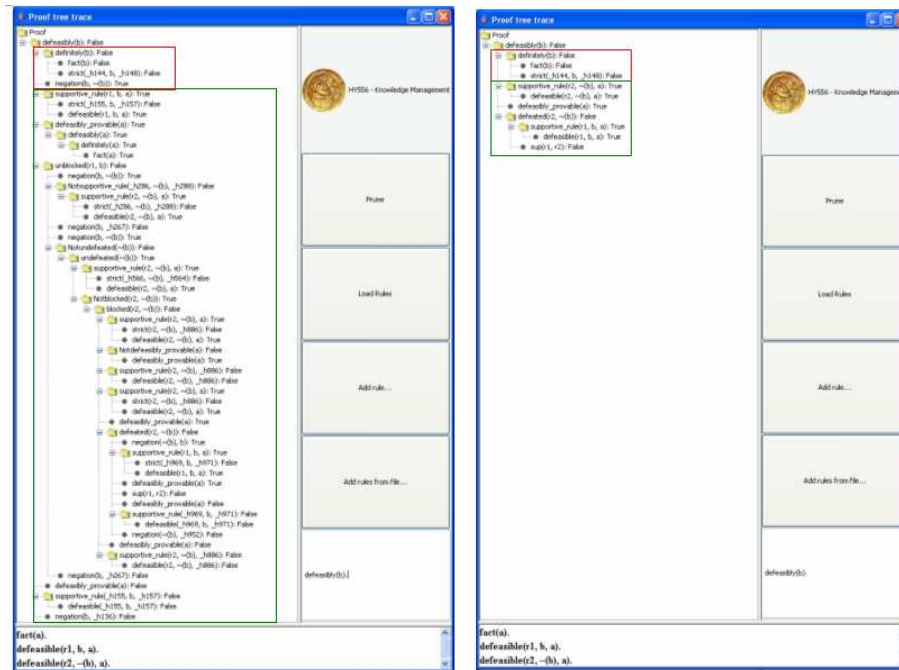


Figure 12: The colored segments correspond to the respective pruned and unpruned versions of the tree. In the unpruned tree, we show all rules that actually fail due to the unblocked attacking rule, whereas in the pruned tree, we just show that rule and the fact that it cannot be defeated

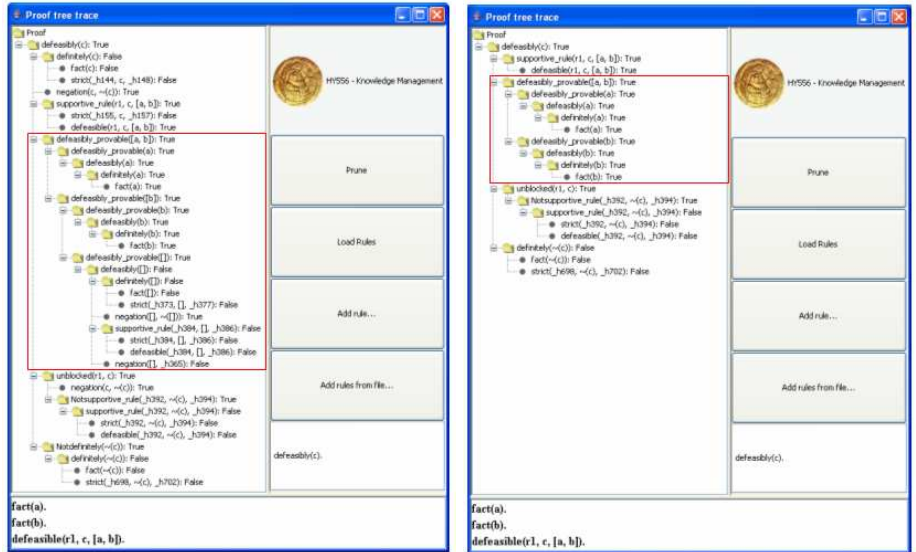


Figure 13: The list pruning can be seen in the colored segments, corresponding respectively to the pruned and unpruned versions. The unpruned tree shows the recursive nature of the Prolog list (a list in prolog is the head element and the remaining sublist), whereas in the pruned tree we simply present the list elements



Figure 14: The handling of the missing proof can be seen in the colored segments, corresponding respectively to the pruned and unpruned versions. We simply clone the specific proof (red) where needed (green)

11 Graphical user interface to the proof system

The graphical user interface 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, indivisible predicates of the proof system. Additionally, if a predicate has additional metadata attached to its definition, such as references for the *fact* predicates, those are displayed as a tooltip to the corresponding tree node.

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 proof tree in order to eliminate the artifacts of the meta-program and simplify its structure (see section 2) or not. Thus, in order 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. By pressing the *OK* button the addition of the given rule is confirmed. Besides that, pressing the *Add rules* from file button allows the user to select a file that contains multiple rules separated by newlines. Those are subsequently added to the system as soon as the file selection is confirmed. The added rules are always visible at the bottom part of the graphical user interface. In any case, a given rule may be prefixed by a string enclosed in square brackets. The given string is then associated with the corresponding rule by the system. This is especially useful for adding references to the supplied facts as those references are displayed as a tooltip when the mouse pointer is over the proof node of the visualization tree that represents the fact. For example the predicate `[http://en.wikipedia.org/wiki/Irony] fact(has_new_buildings(csd))` will have the string `http://en.wikipedia.org/wiki/Irony` attached to it.
2. In order to make the system aware of the added rules, those must be explicitly loaded. The rules that were previously added are loaded by pressing the *Load rules* button at the right part of the graphical user interface.
3. As soon as the rules are loaded, the system is ready to be queried by the user. By typing a ‘question’ 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.
4. By pressing the *Prune* button the system runs the algorithms described in the previous section to eliminate redundant information and metaprogram

artifacts and thus bring the visualized proof tree to a more human friendly form.

12 Agent interface to the proof system

12.1 Architecture

The system makes use of two kinds of agents, the ‘Agent’ which asks questions and the ‘Main Agent’ which is responsible to answer the questions asked. Both agents are based on JADE (Java Agent DEvelopment Framework) which is a software Framework fully implemented in Java language. 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 (which not even need to share the same OS) 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 one, as and when required. JADE is completely implemented in Java language and the minimal system requirement is the version 1.4 of JAVA (the run time environment or the JDK).

Figure 15 shows the process followed by the Main Agent in order to answer a question.

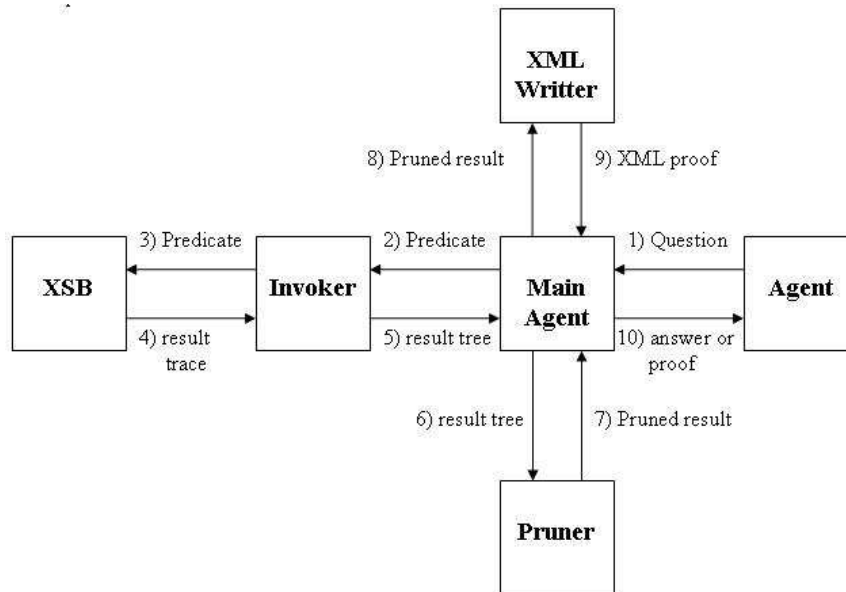


Figure 15: The system architecture

All the above steps are illustrated at the next paragraphs.

1. **An agent asks a Question to the Main Agent.** The question is of the form: **predicate::(proof — answer)**

The predicate must be a valid Prolog predicate, after the predicate must exist two colons (::) and then the word ‘proof’, if the agent wants to take the proof of the predicate, or the word ‘answer’ if the agent wants just the answer to the predicate (true or false). Two examples of questions follow below:

defeasibly(rich(antonis))::proof

defeasibly(rich(antonis))::answer

2. **Main Agent sends the Predicate to the Invoker.** After receiving a question from an agent, the Main Agent has to execute the predicate. For this reason it extracts the predicate from the question 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 MainAgent and sends it to the 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 ‘kind’ of pruning. One is used when the agent that asked the question wants only the result. In that case the tree is pruned and the remaining is just the answer (true or false). The other ‘kind’ of pruning is used when the agent that asked the question wants the proof. In that case, the brunches of the tree that are not needed are pruned, so the remaining is a pruned tree only with brunches 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 asked the question 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 Agents returns Answer or Proof.** Finally the Main Agent sends back to the agent that asked the question 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.

Two kinds of agents that communicate with the Main Agent have been created. One uses a file to read the questions to be asked and the other gets the questions from the user through a Graphical User Interface. These two kinds of agents will be described at the next paragraphs.

12.2 Visual Agent

The visual agent uses a GUI in order to get the questions from the user and send them to the 'Main Agent'. First of all, the user must enter the name of the Responder Agent (called 'Main Agent'), in other words the name of the agent that is going to answer the question. If the name is not entered an error message will appear (Figure 16).

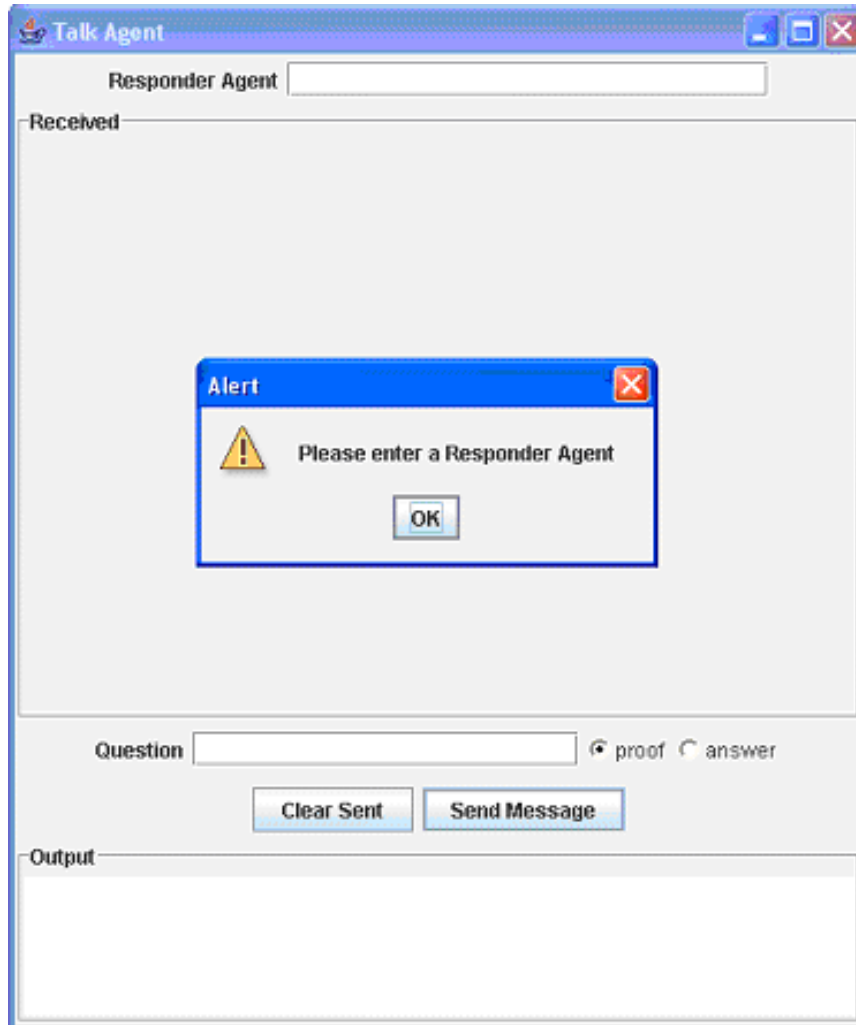


Figure 16: Error message if Responder Agent is not entered

After entering the name of the 'Main Agent' the user must enter the question and chose the type of answer he wants. The question must be a valid Prolog predicate (e.g. `defeasibly(rich(antonis))`). The user can ask for two different types of answer, he can ask just for the answer of the question (true, false), or for the proof of the question. The choice is made though the appropriate radio button. If the question is not entered an error message will appear (Figure 17).

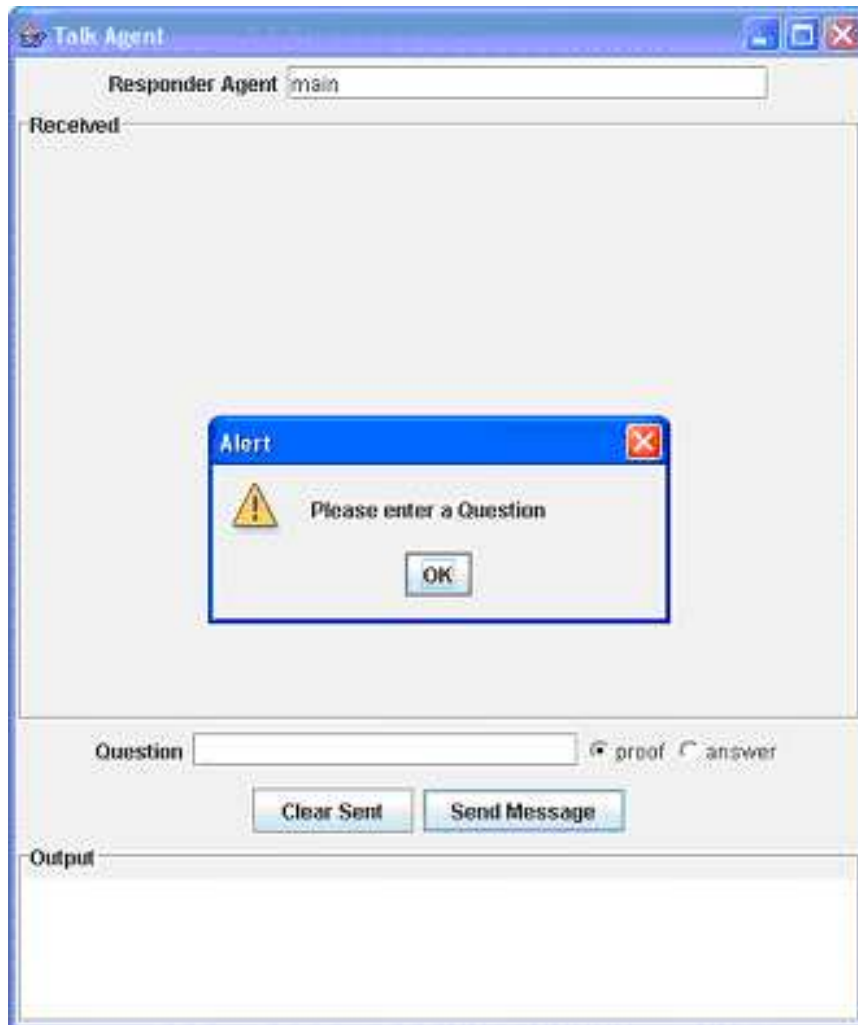


Figure 17: Error message if the question is not entered

If the user chooses to see just the answer of the question, it will appear at the 'Output' area as show at the Figure 18. The answer will be true or false.

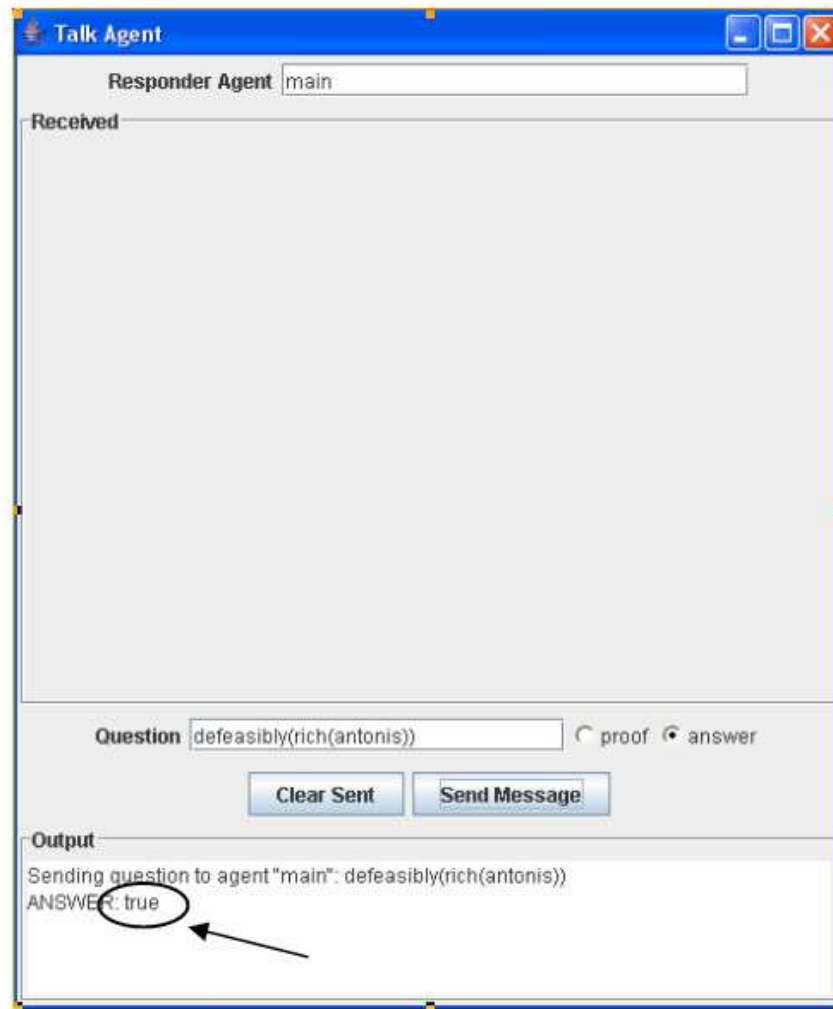


Figure 18: The answer to a question

If the user chooses to see the proof of the question, it will appear at the 'Received' area as show at the Figure 19. The proof is visualized as a tree model. The tree model contains two kind of nodes, the intermediate nodes and the final nodes. The intermediate nodes are represented as folders which can be expanded and show their children nodes. Each intermediate node has the name of a tag used at the XML schema. The final nodes (they can not be more expanded) are represented as files and have a description of their content. The final nodes are also tags of the XML schema.

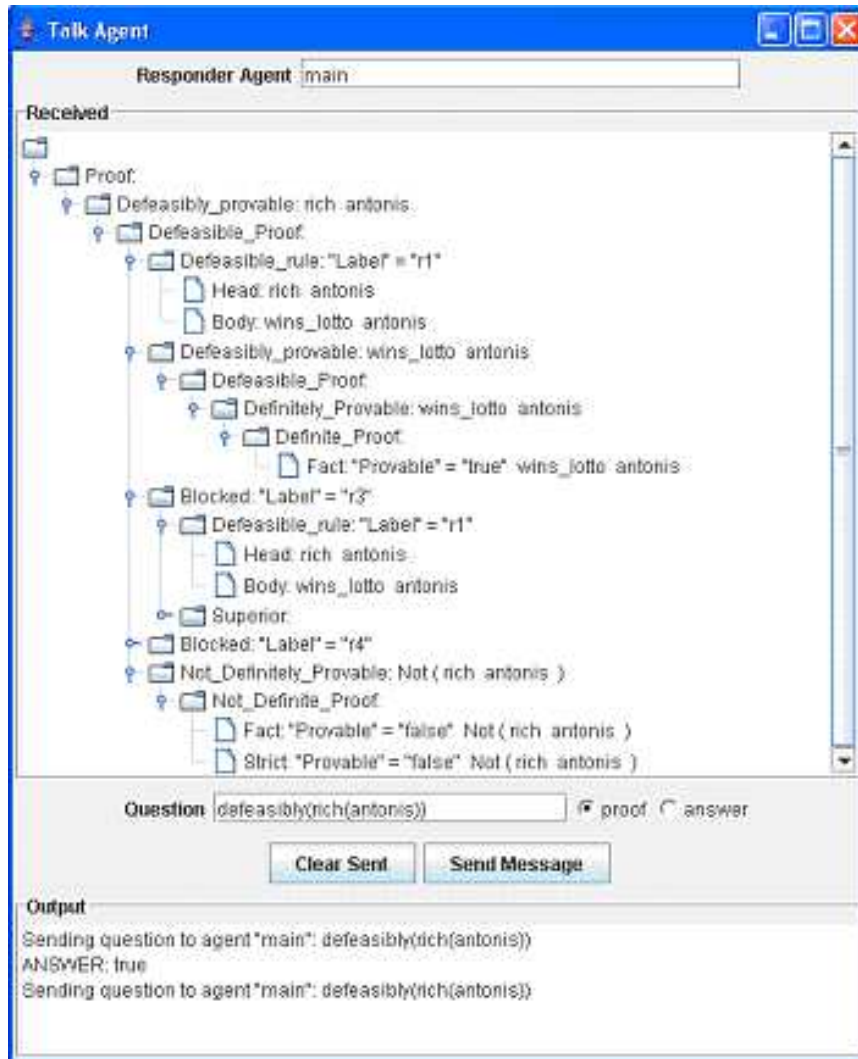


Figure 19: The proof of a question

Finally an error may occur while computing the result. In this case the Main Agent sends an error message to the agent that asked the question. The error message is displayed at the 'Output' area (Figure 20). Many kind of error may occur:

- If the predicate of the question is not defined at the knowledge base of the Main Agent.
- If the question is not syntactically correct (e.g parenthesis missing).
- If the type of answer asked is not valid (proof or answer). This type of error can not occur at the visual agent because this choice is made through a radio button.
- If an exception occurs while computing the result.



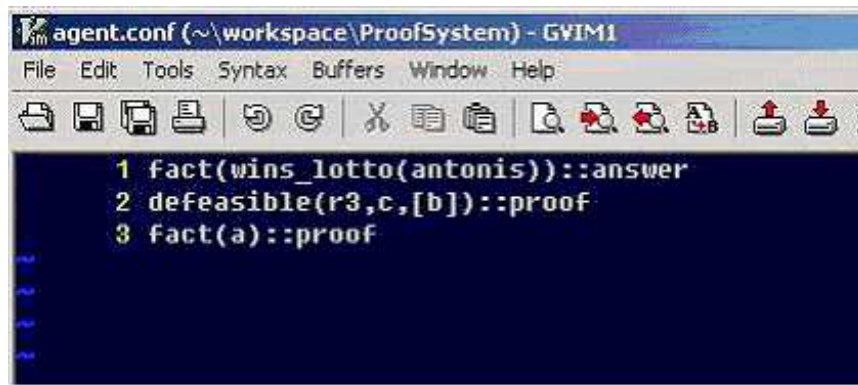
Figure 20: Error messages occur while computing the result

12.3 Command Line Agent

The main goal of the ‘Command Line Agent (CLAgent)’ to make questions to the main Agent and wait for a response, which is an answer or a proof.

12.3.1 Operation of the CLAgent

When the agent starts, it reads, in a random way, the questions it has to make from a configuration file called *agent.conf*. The format of the question is the same as at the visual agent, e.g. *fact (wins_Lotto (antonis))::proof or answer*. The Figure 21 shows a sample configuration file of the Agent.



```
agent.conf (~\workspace\ProofSystem) - GVIM1
File Edit Tools Syntax Buffers Window Help
1 fact(wins_lotto(antonis))::answer
2 defeasible(r3,c,[b])::proof
3 fact(a)::proof
```

Figure 21: Sample configuration file

Then it sends the questions that it has read from the file, to the main Agent. The Figure 22 shows an example communication of the main Agent and the Command Line Agent.



```
31 Iovλ 2006 6:05:05 μμ jade.core.Runtime beginContainer
INFO: -----
This is JADE3.4 - revision 5874 of 2006/03/09 14:13:11
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
-----
31 Iovλ 2006 6:05:06 μμ jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
->Agent main:Waiting for the message...
--Agent test: Sending message....
--Agent test: Sending message....
--Agent test: Sending message....
->Agent main:Received message from agent test@Hacker:1099/JADE
Content of The message: defeasible(r3,c,[b])::proof
```

Figure 22: Communication of the main Agent and the CLAgent

After the main Agent receives the questions from the Agent, then it replies to the CLAgent, sending back the answers or proofs (Figure 23).

```

->Agent main:Creating Reply to agent test@Hacker:1099/JADE
--Agent test: Received message from agent main@Hacker:1099/JADE
Content of The message: PROOF:<?xml version="1.0" encoding="UTF-8"?>
<Proof>
  <Fact Provable="false">
    <Atom>
      <Op>a</Op>
    </Atom>
  </Fact>
</Proof>

->Agent main:Received message from agent test@Hacker:1099/JADE
Content of The message: fact(wins_lotto(antonis))::answer
->Agent main:Creating Reply to agent test@Hacker:1099/JADE
--Agent test: Received message from agent main@Hacker:1099/JADE
Content of The message: ANSWERfalse

Agent test: Terminating

```

Figure 23: Main Agent sends the answer or proof

The kind of errors that can be produced is the same as at the visual Agent. They have the same functionality because the main Agent is the same in the two cases.

Finally the CLAgent keeps a log file called *'name_of_agent_log'*, which contains all the answers and proofs that it has received from the main Agent. An example of the log file is shown in figure 24.

12.4 Agent Use Cases

In this section we mention three examples where the agents make use of the explanation in the Semantic Web.

- An agent can make use of an explanation during a negotiation at E-commerce. 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 represent the online shop answers positively then the buyers agent can ask for an explanation why he owns the money. Then the online shops agent will answer sending back the full explanation.
- An other 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 students grades. Then for every lesson that the student failed to pass the agent may ask for an explanation why he failed. The universitys agent then will respond with a full explanation containing for example the midterms grade the grade of the project and the grade of the final exam. The same can happen for the lessons that the student succeeded where the agents can ask for an explanation how the graded has been extracted.

```
1 #####
2 # AGENT LOG FILE #
3 #####
4
5 QUERIES:
6 1.defeasible(r3,c,[b]::proof
7 2.fact(a)::proof
8 3.fact(wins_lotto(antonis))::answer
9
10 ANSWERS - PROOFS:
11
12 1.PROOF: <?xml version="1.0" encoding="UTF-8"?>
13     <Proof>
14         <Defeasible_rule Label="r3">
15             <Head>
16                 <Atom>
17                     <Op>c</Op>
18                 </Atom>
19             </Head>
20             <Body>
21                 <Atom>
22                     <Op>b</Op>
23                 </Atom>
24             </Body>
25         </Defeasible_rule>
26     </Proof>
27
28 2.PROOF: <?xml version="1.0" encoding="UTF-8"?>
29     <Proof>
30         <Fact Provable="false">
31             <Atom>
32                 <Op>a</Op>
33             </Atom>
34         </Fact>
35     </Proof>
36
37 3.ANSWER: false
```

Figure 24: Log file of the CLAgent

- Finally an agent can ask for an explanation when it not authorised 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 can ask for an explanation why he is not authorised to access the system. An approach in this direction has been developed in the infrastructure in [20]. In this paper is described 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.

At all the above cases the negotiation is made automatically without the users mediation. The agent makes all the appropriate actions and presents only the result - explanation to the user.

13 Relative Work

There are many areas where work has been centered around explanation in reasoning systems. 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 *MYCIN* [14] and *Explainable Expert System* [15], 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* [16,17], which is a knowledge representation language. It was developed a logical infrastructure for separating pieces of logical proofs and automatically generating follow-up questions based on the logical format.

13.1 Inference Web

Nowadays, import work in this area is done by the research group of the *Inference Web* [18]. 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 - if users (humans and agents) are to use and integrate data from unknown, uncertain, or multiple sources, they need provenance metadata for evaluation
- Interoperability - more systems are using varied sources and multiple information manipulation engines, thus increasing interoperability requirements
- Explanation/Justification - if information has been manipulated (i.e., by sound deduction or by heuristic processes), information manipulation trace information should be available
- Trust - if some sources are more trustworthy than others, trust ratings are desired

The Inference Web consists of the following main components:

- *Proof Markup Language (PML)* [19] is an OWL-based specification for documents representing both proofs and proof meta information. Proofs are specified in PML and are interoperable. Proof fragments as well as entire proofs may be combined and interchanged. So PML provides the Inference Web 's support for distributed proofs.
- *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. By providing meta information for sources,

the Inference Web supports knowledge provenance. It also supports reasoning information, which is provided by the PML documents and the IWBase, which supports meta information about inference engines along with their primitive inference rules.

- *IWExplainer* is a tool for abstracting proofs into more understandable formats. It supports various strategies to explain answers including the visualization of abstracted proof, presentation of provenance information, etc.
- *IWBrowser* can display both proofs and explanations in number of proof styles and sentence formats.

Beyond just explaining a single system, Inference Web attempts to provide a way of combining and presenting proofs that are available. It does not take one stance on the form of the explanation since it allows deductive engines to dump single or multiple explanations of any deduction in the deductive language of their choice. It provides the user with flexibility in viewing fragments of single or multiple explanations in multiple formats. IW simple 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 question answering system must register in the IWBase its inference engine along with its supported inference rules, using the PML specification format. The IW supports proof generation service that facilitates the creation of PML proofs by inference engines.

Inference Web was originally aimed at explaining answers from theorem provers that encode a set of declaratively specified inference rules. Theorem provers like Stanford's JTP reasoner and SRI's SNARK reasoner have been registered and they produce PML proofs. Prototype implementations of Semantic Web agents that are based on the JTP theorem prover are supported by the IW. Future work includes the registration of more question answering systems from different areas, like query planners and extractors.

It is an interesting and open issue if 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 that produced by our proof system.

Extra work needs to be done in Inference Web in order to support why-not questions. Current IW infrastructure can not 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.

14 Conclusions and Future Work

This work resulted to a new system that aims to increase the trust of the users for the Semantic Web applications. We created a system that automatically generate an explanation for every answer to the users questions, in a formal and useful representation. This system can be used by individual users that want to get a more detailed explanation from a reasoning system in the Semantic Web, in a more human readable way. Our reasoning system was based on defeasible logic and we used the relative implemented meta-program, where XSB was used as the reasoning engine. We developed a pruning algorithm that reads the XSB's trace and removes the redundant information in order to formulate a sensible proof. Furthermore, the system can be used by agents that is common in many applications in the Semantic Web. An other contribution of our work is a new XML language 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.

Besides the current implementation, there is much future work that can be done. Also, much improvement can be achieved for a more human friendly presentation of a logical explanation (without assuming any knowledge for the underlying logical system). Our XML-based language for explanation representation is not fully compatible with RuleML (e.g an extention of RuleML 's specification has been developed for situated courteous logic programs). This is a possible extension of our XML schema. Finally, our system could be integrated to the Inference Web framework, by registering our inference engine and inference rules and converting our representation in the PML format.

References

- [1] T. Berners-Lee (1999). Weaving the Web. Harper 1999.
- [2] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler (2000). Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation, October 2000. Available at: <http://www.w3.org/TR/2000/RECxml-20001006>. G. Antoniou and F. van
- [3] G. Antoniou and F. van Harmelen (2004). A Semantic Web Primer. MIT Press 2004.
- [4] D.L. McGuinness , F. van Harmelen (2004). OWL Web Ontology Language Overview W3C Recommendation, February 2004. Available at: <http://www.w3.org/TR/owl-features/>.

- [5] G. Antoniou, D. Billington, G. Governatori and M.J. Maher (2001). Representation results for defeasible logic. *ACM Transactions on Computational Logic* 2, 2 (2001): 255–287.
- [6] B. N. Grosz (1997). Prioritized conflict handling for logic programs. In *Proc. of the 1997 International Symposium on Logic Programming*, 197-211.
- [7] G. Antoniou, M. J. Maher and D. Billington (2000). Defeasible Logic versus Logic Programming without Negation as Failure. *Journal of Logic Programming* 41,1 (2000): 45–57. M.J. Maher (2002).
- [8] A Model-Theoretic Semantics for Defeasible Logic, *Proc. Workshop on Paraconsistent Computational Logic*, 67 - 80, 2002.
- [9] D. Beckett (2004). *RDF/XML Syntax Specification*, W3C Recommendation, February 2004. Available at: <http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/>.
- [10] D. Brickley, R.V. Guha (2004). *RDF Vocabulary Description Language 1.0: RDF Schema* W3C Recommendation, February 2004. Available at: <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [11] Antonis Bikakis, Grigoris Antoniou: *DR-Prolog: A System for Reasoning with Rules and Ontologies on the Semantic Web*. *AAAI 2005*: 1594-1595
- [12] Nick Bassiliades, Grigoris Antoniou, Ioannis P. Vlahavas: *DR-DEVICE: A Defeasible Logic System for the Semantic Web*. *PPSWR 2004*: 134-148
- [13] RuleML. The Rule Markup Language Initiative. www.ruleml.org
- [14] E. Shortliffe. *Computer-based Medical Consultations: MYCIN*. Elsevier, 1976.
- [15] W. Swartout, C. Paris, and J. Moore. Design for explainable expert systems. *IEEE Expert*, 6(3):58–647, 1991.
- [16] Deborah L. McGuinness and Alex Borgida. Explaining Subsumption in Description Logics, in *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, August 1995.
- [17] Deborah L. McGuinness. *Explaining Reasoning in Description Logics*, Rutgers University Thesis, New Brunswick, 1996.
- [18] Inference Web. *Semantic Web Infrastructure for provenance and justification*. <http://iw.stanford.edu/2.0/>
- [19] Paulo Pinheiro da Silva, Deborah L. McGuinness and Richard Fikes. *A Proof Markup Language for Semantic Web Services*. *Information Systems*. Volume 31, Issues 4-5, June-July 2006, Pages 381-395. Previous version, technical report, Knowledge Systems Laboratory, Stanford University.

A XML Schema for Explanation Representation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "Definitely_provable" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "Atom" />
        <xsd:element ref = "Definite_Proof" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "Atom">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name= "Op"/>
          <xsd:sequence minOccurs = "0" maxOccurs = "unbounded">
            <xsd:element name= "Var" minOccurs = "0"/>
            <xsd:element name = "Ind" minOccurs = "0"/>
          </xsd:sequence>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="Not">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name= "Op"/>
                <xsd:sequence minOccurs = "0" maxOccurs = "unbounded">
                  <xsd:element name= "Var" minOccurs = "0"/>
                  <xsd:element name = "Ind" minOccurs = "0"/>
                </xsd:sequence>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "Definite_Proof">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element ref= "Strict_rule"/>
          <xsd:element ref= "Definitely_provable" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:element ref= "Fact"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "Strict_rule">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref= "Head"/>
        <xsd:element ref= "Body"/>
      </xsd:sequence>
      <xsd:attribute name = "Label" type = "xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name= "Head">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref= "Atom"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name= "Body">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref= "Atom" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name= "Fact">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref= "Atom" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "Defeasibly_provable" >
    <xsd:complexType>
        <xsd:choice>
            <xsd:element ref="Definitely_provable" />
            <xsd:sequence>
                <xsd:element ref = "Atom" />
                <xsd:element ref = "Defeasible_Proof" />
            </xsd:sequence>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
<xsd:element name = "Defeasible_Proof">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:choice>
                <xsd:element ref= "Strict_rule" />
                <xsd:element ref= "Defeasible_rule" />
            </xsd:choice>
            <xsd:element ref="Defeasibly_provable" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element ref="Not_Definitely_provable" />
            <xsd:element ref="Blocked" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name = "Defeasible_rule">
    <xsd:complexType>
        <xsd:sequence minOccurs="0">
            <xsd:element ref= "Head" />
            <xsd:element ref= "Body" />
        </xsd:sequence>
        <xsd:attribute name = "Label" type ="xsd:string" use="optional" />
    </xsd:complexType>
</xsd:element>
<xsd:element name = "Blocked">
    <xsd:complexType>
        <xsd:choice>
            <xsd:sequence>
                <xsd:element ref="Defeasible_rule" />
            <xsd:choice>
                <xsd:element ref="Superior" />
                <xsd:element ref="Not_Defeasibly_provable" />
            </xsd:choice>
        </xsd:sequence>
    <xsd:sequence>

```

```

        <xsd:element ref="Strict_rule"/>
        <xsd:element ref= "Not_Definitely_provable"/>
    </xsd:sequence>
    <xsd:element name="Not_Superior">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Defeasible_rule"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:choice>
</xsd:complexType>
</xsd:element>
<xsd:element name = "Superior">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="Defeasible_rule"/>
            <xsd:element ref="Defeasibly_provable" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "Not_Definitely_provable">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "Atom" />
            <xsd:element ref = "Not_Definite_Proof" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name = "Not_Definite_Proof">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref= "Blocked" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "Not_Defeasibly_provable">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "Atom" />
            <xsd:element ref = "Not_Defeasibly_Proof" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name = "Not_Defeasibly_Proof">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref= "Not_Definitely_provable"/>
            <xsd:choice>
                <xsd:element ref= "Blocked" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="Definitely_provable"/>
                <xsd:element ref="Undeclared"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="Undeclared">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref= "Defeasible_rule"/>

```

```
        <xsd:element ref="Defeasibly_provable" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="Blocked" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```