University of Crete
School of Sciences and Engineering
Computer Science Department

A System for Modal and Deontic Defeasible Reasoning

by

Nikos Dimaresis

Master 's thesis

Heraklion, July 2007

Computer Science Department
School of Sciences and Engineering
University of Crete

A System for Modal and Deontic Defeasible Reasoning

Submitted to the Department of Computer Science
in partial fulfillment of the requirements
for the degree of Master of Science

Author: _____
Nikos Dimaresis
Computer Science Department

Board of Inquiry:

Supervisor: _____
Grigoris Antoniou, Professor

Member: _____
Dimitris Plexousakis, Professor

Member: _____
Anastasia Analyti, Researcher

Accepted by: _____
Panagiotis Trahanias, Professor
Chairman of the Graduate Studies Committee

July 2007, Heraklion, Greece

A System for Modal and Deontic Defeasible Reasoning

Dimaresis Nikos

Master of Science Thesis

Computer Science Department, University of Crete

Abstract

Defeasible logic is a simple and efficient rule-based nonmonotonic reasoning approach, that has been shown useful for various applications areas. Recently defeasible logic has been used in applications to the Semantic Web.

The Semantic Web is an extension of the current Web, in which information is given well-defined meaning, and its development proceeds in layers, each layer being on top of other layers. Now that the layers of metadata (RDF) and ontology (OWL) have reached sufficient maturity, the next step will be the logic and proof layer and an important focus is on rule languages for the Semantic Web. While initially the focus has been on monotonic rule systems, nonmonotonic rule systems are increasingly gaining attention.

The first source of motivation for our work is the modelling of multi-agent systems based on cognitive and social models, where an agent behavior is determined as an interplay between mental attitudes and normative aspects. Commonly, these aspects are logically captured through the use of modal logics, which are by definition monotonic. Reasoning about intentions and other mental attitudes has defeasible nature, and defeasibility is a key aspect for normative reasoning.

The second important source for motivation for our work is the modelling of policies. Defeasible logic is the nonmonotic reasoning approach that would be suitable solution for the requirements that arise from the specific nature of policies and especially of business rules.

In our work we use and develop an extend variant of defeasible logic, that uses modal and deontic operators. This is a suitable formalism that can deal with the motivational components of our work and can capture their nonmonotonic behavior. For the purposes of modelling policies sufficiently, we will introduce an additional deontic operator for expressing "permission".

We implement a nonmonotonic rule based system, based on this formalism, which integrates with the Semantic Web, as it reasons with the standards of RDF and RDF Schema. The core of the system consists of a logic metaprogram that implements the extension of defeasible logic.

"Ενα Σύστημα για Δεοντική και Τροπική Συλλογιστική"

Νίκος Δημαρέσης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Η Αναιρέσιμη Λογική είναι μία απλή και αποδοτική βασισμένη-σε-κανόνες μη μονοτονική προσέγγιση, η οποία έχει αποδειχτεί χρήσιμη σε διάφορες περιοχές εφαρμογών. Πρόσφατα η Αναιρέσιμη Λογική έχει χρησιμοποιηθεί σε εφαρμογές πάνω στο Σημασιολογικό Ιστό.

Ο Σημασιολογικός Ιστός είναι μία επέκταση του σημερινού Ιστού, όπου η πληροφορία έχει καλά καθορισμένο νόημα, και η ανάπτυξη του βασίζεται σε πρότυπα γλωσσών δομημένων σε επίπεδα, που κάθε επίπεδο βρίσκεται πάνω από άλλα επίπεδα. Τώρα που τα επίπεδα των μετα-δεδομένων (RDF) και οντολογιών (OWL) έχουν αναπτυχθεί επαρκώς, το επόμενο βήμα είναι η ανάπτυξη των επιπέδων της λογικής και της τεκμηρίωσης. Τα τελευταία χρόνια η έρευνα επικεντρώνεται στις γλώσσες κανόνων για το Σημασιολογικό Ιστό. Ενώ η αρχική έρευνα εστίασε πάνω στα μονοτονικά συστήματα κανόνων, τα μη μονοτονικά συστήματα κανόνων κερδίζουν όλο και περισσότερη προσοχή.

Η πρώτη πηγή παρακίνησης για τη δουλειά μας είναι η μοντελοποίηση πολυ-πρακτορικών συστημάτων, βασισμένα πάνω σε γνωστικά και κοινωνικά μοντέλα, στα οποία η συμπεριφορά ενός πράκτορα καθορίζεται από την αλληλεπίδραση μεταξύ διανοητικών διαθέσεων και κανονιστικών όψεων.

Η δεύτερη σημαντική πηγή παρακίνησης για την δουλειά μας είναι η μοντελοποίηση πολιτικών. Η Αναιρέσιμη Λογική είναι η προσέγγιση μη μονοτονικής συλλογιστικής που θα ήταν η κατάλληλη λύση για τις απαιτήσεις που εμφανίζονται λόγω της συγκεκριμένης φύσης των πολιτικών και ειδικότερα των επιχειρησιακών κανόνων.

Στην εργασία μας χρησιμοποιούμε και αναπτύσσουμε μία επεκτάσιμη παραλλαγή της Αναιρέσιμης Λογικής, η οποία χρησιμοποιεί Τροπικούς και Δεοντικούς τελεστές. Αυτή είναι μία κατάλληλη τυπική γλώσσα που μπορεί να

χειριστεί τα συστατικά παρακίνησης της δουλειάς μας και να συλλάβει τη μη μονοτονική συμπεριφορά τους. Για τους σκοπούς μοντελοποίησης πολιτικών επαρκώς, θα εισάγουμε ένα επιπλέον δεοντολογικό τελεστή για να εκφράσουμε την έννοια της "άδειας".

Υλοποιήσαμε ένα μη μονοτονικό βασισμένο-σε-κανόνες σύστημα, στηριζόμενο σε αυτή τη τυπική γλώσσα, το οποίο αλληλεπιδρά με τα πρότυπα RDF και RDF Schema του Σημασιολογικού Ιστού. Ο πυρήνας του συστήματος αποτελείται από ένα λογικό μεταπρόγραμμα, το οποίο υλοποιεί την επέκταση της Αναιρέσιμης Λογικής.

Ευχαριστίες

Αισθάνομαι την ανάγκη να ευχαριστήσω τον επόπτη καθηγητή μου, κύριο Γρηγόρη Αντωνίου, για την πολύτιμη καθοδήγησή του και την ουσιαστική συμβουλή του στην ολοκλήρωση της εργασίας, αλλά και για την αποδοτική και ευχάριστη συνεργασία που είχαμε.

Θα ήθελα να ευχαριστήσω την κυρία Αναστασία Αναλυτή, ερευνήτρια του ΙΤΕ-ΙΠ, για τις πολύτιμες συμβουλές και παρατηρήσεις της, και για την συμμετοχή της στην εισηγητική επιτροπή της εργασίας, και τον καθηγητή Δημήτρη Πλεξουσάκη για την συμμετοχή του στην εισηγητική επιτροπή.

Επιπλέον, ευχαριστώ τον διδακτορικο φοιτητή Αντώνη Μπικάκη, για την συμβολή του στην επίλυση πλήθους αποριών μου και για την αξιόλογη προθυμία του για βοήθεια.

Τέλος ευχαριστώ όλους τους φίλους μου για όλες τις στιγμές που περάσαμε μαζί και για όλη την βοήθεια που μου πρόσφεραν κατά διάρκεια των σπουδών μου στο Ηράκλειο της Κρήτης.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first source of motivation for our work is the modelling of multi-agent systems, in which agents can operate effectively and interact with each other productively. In particular, we follow more recent approaches on cognitive agents that combine two apparently independent perspectives: (a) a cognitive account of agents that specifies motivational attitudes, and (b) modelling of agent societies by means of normative concepts. The first aspect is addressed through the well-known BDI architecture [18, 71]. The second aspect is based on artificial societies of agents, in which normative concepts play a decisive role, allowing for coordination of autonomous agents [31, 69]. The result of this combination of perspectives is the modelling of autonomous agents based on cognitive and social models, where an agent deliberation and behavior is determined as an interplay between mental attitudes and normative aspects.

Commonly, both motivational attitudes and normative aspects are logically captured through the use of modal logics. Modal logics are extensions of classical propositional logic with some intensional operators. So modal logics are by definition monotonic. However as we know, classical propositional logic is not well suited to deal with real life scenarios and inconsistent information, that may easily arise in multi-agent and web environments. As argued in [40], reasoning about intentions and other mental attitudes has *defeasible nature*, and defeasibility is a key aspect for normative reasoning.

The second important source of motivation for our work is the *modelling of policies*. Policies play crucial roles in enhancing security, privacy, and usability of distributed services and extensive research has been done in this area, including the Semantic Web community [16]. It encompasses the notions of security policies, trust management, action languages, and business rules. Business rules are statements that are used by a body or an organization to run their activities. They provide a foundation for understanding how a business operates. They are used to formalize and automate business decisions as well as for efficiency reasons.

As explained in [1], defeasible reasoning is appropriate for modelling and reasoning with business rules. However, in order to be able to represent and reason

with business rules sufficiently, there are still requirements which go beyond defeasible logic. In particular, we need a formal specification language with higher expressiveness, including deontic notions[81].

In our work, we adopt the well-known defeasible logic, that is described in [5], as the suitable formalism that can deal with these components and capture their nonmonotonic behavior. Defeasible logic has been studied in terms of proof theory [5], model-theoretic semantics [57], and argumentation semantics [37], and has delivered efficient implementations [2, 59]. It is a flexible, rule-based, and efficient approach, that has been shown useful for application areas, such as modelling of contracts [46, 44, 35], legal reasoning [41], agent negotiations [36], modelling of agents and agent societies [40, 38], and applications to the Semantic Web [2, 12]. Recent work shows that defeasible logic is a nonmonotonic approach that can be extended with modal and deontic operators [40], [39], [41], [74], [38].

This thesis presents a computationally-oriented nonmonotonic logical framework, based on the approach of [40], that extends defeasible logic with modal and deontic operators, and reports on an implemented system, based on this formalism. This proposed logic introduces and manipulates modalities, and is flexible enough to deal with different intuitions about the interactions of the internal and external motivational attitudes.

As stated, the expressive power of the formal specification language that is required by the business rules community is high and includes deontic notions like obligation, permission, and prohibition. This task is captured by the deontic extension of defeasible logic. For the purposes of modelling policies, we introduce an additional deontic operator to our logical framework, in order to express *permission*. This operator is used commonly in policies, describing (conditional) entitlements.

The Semantic Web is an extension of the current Web, in which information is given a well-defined meaning, better enabling computers and people to work in cooperation. The development of the Semantic Web proceeds in layers, each layer being on top of other layers. Now that the layers of metadata (RDF) and ontology (OWL) are stable, an important focus is on rule languages for the Semantic Web. While initially the focus has been on monotonic rule systems [42, 47, 75], nonmonotonic rule systems are increasingly gaining attention [29, 12, 2]. Our language of choice, defeasible logic, is compatible with applications in this area. In particular, there are implementations of defeasible logic that interoperate with Semantic Web standards [12, 2]. The two motivations of our work outlined above can be combined with the *Semantic Web* initiative [13], as Semantic web languages and technologies support the issue of *semantic interoperability*, which is important both for multi-agent systems and for policies.

As already stated, this thesis presents modal and deontic extensions of defeasible logic, and describe an implemented system, the basic characteristics of which are the following:

- It is a nonmonotonic rule-based system that supports reasoning in defeasible

logic, extended with modalities.

- It integrates with the Semantic Web, as it reasons with the standards of RDF and RDF Schema.
- It is based on Prolog. The core of the system consists of a logic metaprogram that implements the extension of defeasible logic. In particular, we base our implementation on the system DR-Prolog [2], which uses XSB [91] as the underlying logical engine.

This rest of the thesis is organized as follows:

Chapter 2 presents the Semantic Web and the role of rules in its development. Firstly, we present the layers of the Semantic Web that have been so far implemented. At present, the highest layer that has reached sufficient maturity is the ontology layer in the form of the description logic based languages. Then we reason why rule systems, especially the nonmonotonic ones, are expected to be part of the layered development of the Semantic Web, for the realization of logic and proof layers. We report on rule systems, both monotonic and nonmonotonic, as the approaches in integrating ontologies with rules is a subject of active research for the Semantic Web community. Finally, we present rule languages, as standardization efforts in representing rules for the Semantic Web.

Chapter 3 presents the logical formalism. At first, we present the language of defeasible logic and its main features and then one motivation of our work, the multi-agent systems. We outline different perspectives, like agents based on cognitive and social models and approaches on this domain, that include BDI and BOID architecture. After presenting modal logics and its variations, we describe why motivational attitudes and normative notions have nonmonotonic behavior. Thus we conclude that extending defeasible logic with modal and deontic operators as a convenient and appropriate way to model these aspects.

Chapter 4 presents the logic metaprogram, implemented in Prolog, that was used to implement the extension of defeasible logic. We present the predicates and the clauses that consist the different parts of the metaprogram, in order to formulate the defeasible theory of the formalism. We show the different ways that can be used to handle the additional operator of permission and the resolution of conflicts among modalities for the different types of agents. Finally, we illustrate with several examples how we use this metaprogram in order to reason over this formalism.

Chapter 5 reports on the implementation architecture of our nonmonotonic rule-based system, based on the extension of defeasible logic. Firstly, we give an overview of how the system works, and then we describe in detail the functionality of each of the modules of the system. We show with several screenshots the GUI of this system, illustrating the way a user interacts with the underlying system.

In Chapter 6 we present a use case, showing in practice the abilities and the functionality of our system. This is an example from a specific application, the modelling of a variety of university regulations from the Department of Computer

Science at the University of Crete. This task is compatible with the motivation of our work in modelling policies and business rules. We use our formalism, in particular the deontic extensions of defeasible logic, to model logically and represent these regulations. RDF/S data are loaded as ontological knowledge, integrating in this way with Semantic Web standards. Then, we show through several screenshots how the system responses to user queries.

Finally, in Chapter 7 we present our conclusions and plans of future work.

Chapter 2

Rules for the Semantic Web

The Semantic Web [13] is an initiative that aims at improving the current state of the World Wide Web. It is an evolving extension of the World Wide Web in which Web content can be expressed not only in natural language, but also in a form that is more easily machine-processable. We can take advantage of these representations by using intelligent techniques. The Semantic Web is propagated by the World Wide Web Consortium (W3C), an international standardization body for the Web. The driving force of the Semantic Web initiative is Tim Berners-Lee, the very person who invented the WWW in the late 1980s. His vision for the Semantic Web is to augment the existing Web with resources more easily interpreted and used by programs and intelligent software agents. This involves moving the Web to a universal medium for data, information, and knowledge exchange. In order to achieve these goals, a variety of enabling technologies are necessary, that will lead to a more advanced Semantic Web. The key technologies include explicit metadata, ontologies, logic and inferencing, and intelligent agents.

The development of the Semantic Web proceeds in steps, each step building a layer on top of another. In building one layer of the Semantic Web on top of another requires each layer to have downward compatibility and upward partial understanding. Downward compatibility means that agents which are fully aware of a layer, should also be able to interpret and use information written at lower levels. Upward partial understanding means that agents should take at least partial advantage of information at higher levels. For example, an agent aware only of the RDF and RDF Schema semantics can interpret knowledge written in OWL partly, by disregarding all but RDF and RDF Schema elements. The Semantic Web layers are presented in Figure 2.1. In the next sections we will briefly describe the basic Semantic Web layers and the technologies that have reached a reasonable degree of maturity.

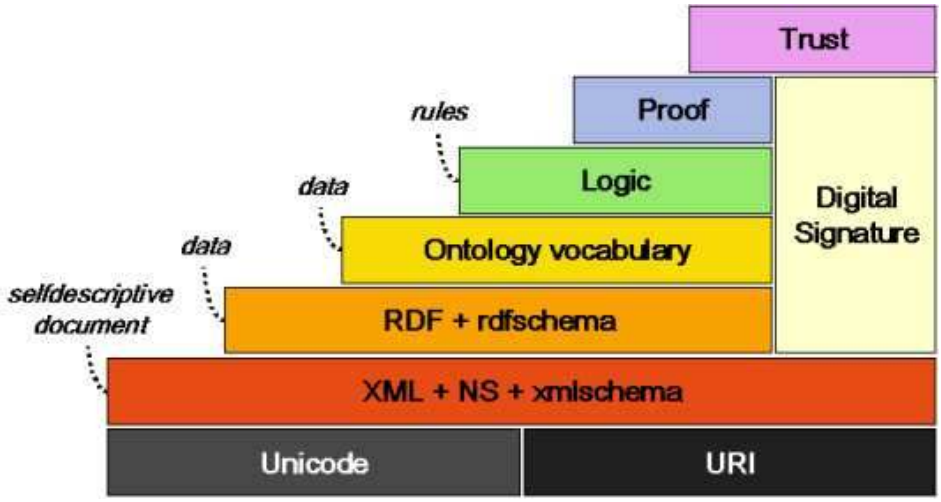

Figure 2.1: The Semantic Web layers from W3C.

2.1 Web Documents in XML

At the bottom layer we find XML [20], a universal meta-language that lets users write structured Web documents with a user-defined vocabulary. The eXtensible Markup Language (XML) provides a uniform framework for exchanging data between applications through markup, structure, and transformations. It represents structured information on the Web, that can be easily accessed by the machines.

Unicode characters and Uniform Resource Identifiers (URIs) are two technologies that XML is built upon. The Unicode characters allow XML information to be communicated using any written human language. URI is a generic term that identifies a resource on the World Wide Web. So URIs are used by the upper levels as unique identifiers for concepts in the Semantic Web.

An XML document is valid if it conforms to certain structuring information. XML Schema [85] is the richer language for defining restrictions in the structure of XML documents. It supports powerful capabilities in defining user-defined data types. It also provides a sophisticated large set of built-in data types, which many of them can be used by the ontology languages in the upper levels.

Namespaces [19] are a simple mechanism of XML for creating globally unique names for the elements and attributes in an XML instance. They are used for disambiguation purposes. An XML document may contain element or attribute names from different markup languages, allowing to be mixed together without ambiguity. A namespace is denoted as a URI reference, which is the location of a vocabulary (e.g an XML Schema).

XML provides a surface syntax for structured documents and Semantic Web technologies, which are positioned in the upper levels and are built on top of this language. Although it provides syntactic interoperability, XML does not provide a mechanism to deal with the the meaning of the content. For this lack of semantics,

other technologies with further features for the Semantic Web are layered on top of XML.

2.2 Describing Web Resources in RDF

XML is a markup language that can often add meaning to data with the use of tags. However, there is no standard way of assigning meaning to tag nesting, actually understanding is meaningful only to humans. It is required for machines to do more automatically and go beyond the notion of XML data model, toward a more meaning model. These machine-processing capabilities are provided by RDF, a foundation for representing and processing metadata. Metadata is the term that captures the information of the data and is used to identify and extract information from Web sources in the Semantic Web.

Resource Description Framework(RDF) [52] is a data-model that is based upon the idea of subject-predicate-object triple, called a *statement*. These statements are made about resources that can be anything with an associated URI. The basic RDF graph-based model produces triples, where a resource (subject) is linked through an arc, labeled with a property (predicate), to a value (object). Properties are special kind of resources (so they are identified by a URI) and they are used to describe relationships between resources. A value can be either a resource or a *literal*, which is an atomic value.

In RDF we can interpret a statement in three ways: a) as a triple, b) as a graph, or c) in a XML form. An example of statement is the sentence:

The course with homepage http://www.csd.uoc.gr/∼hy467 is taught by Grigoris Antoniou.

The simplest way of representing this statement is as a triple:

(*http://www.csd.uoc.gr/∼hy467, http://www.mydomain.org/uni-ns/#isTaughtBy,* "*Grigoris Antoniou*").

Figure 2.2 shows the second graph-based way of representing statements. It is the corresponding graph for the same sentence. We can think that a graph rep-

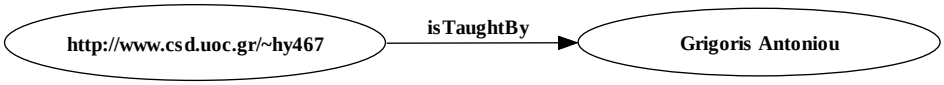

Figure 2.2: Graph representation of an RDF Statement.

resents a collection of interrelated statements, where the nodes are connected via various relationships (properties). At any point we can introduce new nodes, as we add relationships between resources. The following statement

The course with homepage http://www.csd.uoc.gr/~hy467 it entitled as Knowledge Representation.

is added to the previous graph:

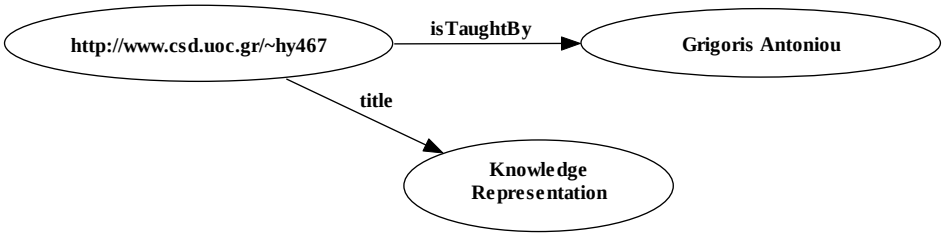

Figure 2.3: RDF graph

Although the graph view is the most convenient for communication between people, the Semantic Web vision requires a suitable representation that is machine-accessible and machine-processable. This is the third way, the XML-based syntax. RDF is defined as an excellent complement to XML. The expression of RDF data in XML form provides syntactic interoperability. It can passed over the Web and be easily accessed by the machines. RDF provides the semantic interoperability and makes an information object interoperable among applications. This combination enables users or machines to retrieve, process and manage information from the Semantic Web.

According to the XML-based view, an RDF document is represented by an XML element, which describes a number of statements about resources.The following RDF/XML document describes the previous statements:

```
<?xml version="1.0" encoding="UTF-16"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:mydomain="http://www.mydomain.org/uni-ns">
     <rdf:Description rdf:about="http://www.csd.uoc.gr/~hy467">
       <mydomain:isTaughtBy rdf:resource="Grigoris Antoniou"/>
       <mydomain:title rdf:datatype="&xsd;string">
            Knowledge Representation
       </mydomain:title>
     </rdf:Description>
 </rdf:RDF>
```

In the first line an RDF document specifies that we are using XML. It follows with the root element tag *rdf:RDF* that also specifies a number of namespaces. The namespace mechanism of XML is used, but in an expanded way. In XML, namespaces are only used to remove ambiguities, while in RDF, external namespaces are expected to be RDF documents defining resources, which are used to import RDF documents. This allows reuse of resources and enables other people to add additional features for the resources producing a large distributed collection of knowledge.

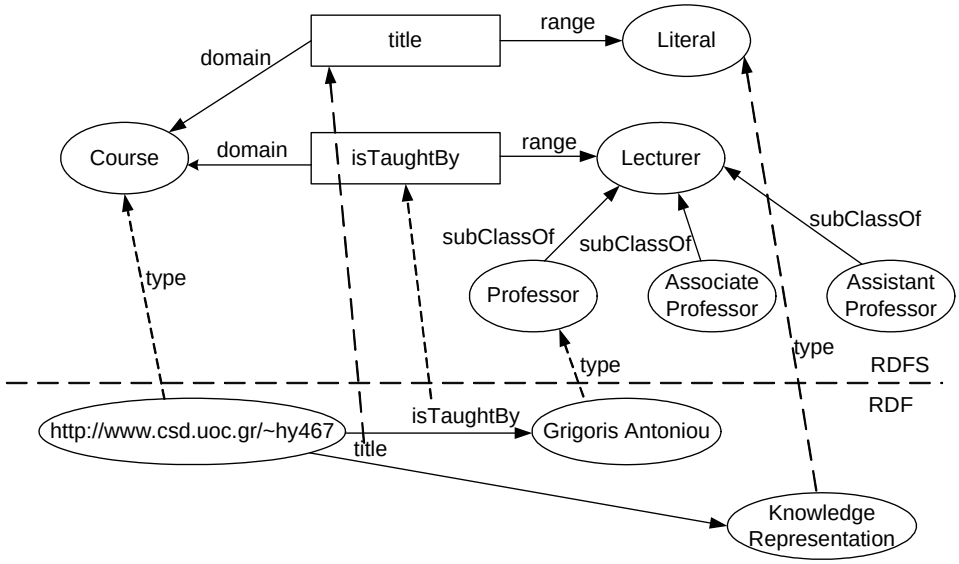

Figure 2.4: RDF Schema and RDF Layers

The content of the rdf:RDF element is a number of statements about resources. We use the element *rdf:Description* to describe each statement about resources. Within the rdf:Description are contained the property elements, the content of which are their values, if the value is a literal. In case the value is a resourse, is denoted as an attribute of a property element. RDF also uses the predefined set of XML Schema data types in order to indicate the type of a literal.

RDF is a universal language that lets users describe resources, but it is domain-independent. That means that it does not make assumptions about any particular application domain. It is up to users to define their own vocabularies using a language called *RDF Schema* (*RDFS*)[21].

RDF Schema is a primitive ontology language that provides a mechanism for describing specific domains. The purpose of RDFS is to provide a vocabulary that specifies which properties apply to which kinds of objects and what values they can take, and describe the relationships between objects. Thus RDFS describes the semantics and makes them machine-accesible, in accordance with the Semantic Web vision. The main RDFS constructs are classes, subclass relations, properties, subproperty relations, and domain and range restrictions.

A class can be considered as a set of elements. Individual objects that belong to a class are instances of this class. The constraints on properties are introduced through domain and range restrictions. The domain specifies the set of resources that may have a given property, while the range specifies the set of values for a given property. Finally RDFS defines hierarchy for classes and properties and establishes these relationships through subclasses and subproperties respectively.

Figure 2.4 shows an example that illustrates the separate layers of RDF Schema and RDF. This schema contains the classes *course* and *lecture*, which it has the

subclasses *professor*, *associate professor* and *assistant professor* and the properties *is taught by* and *title*. It also defines the domain and range restrictions for the properties. The RDF layer contains class instances and values of properties. The arc with label *type* defines the relationship between classes and instances.

Although RDF and RDFS form a basis on which more layers can be built, together they still lacked sufficient expressive power. For example, a serious restriction about RDF is that it uses only binary properties, because we often use predicates with two or more arguments. RDF Schema is a minimal ontology modeling language for the Web. Many desirable modelling primitives are missing and the only constraints expressible are domain and range constraints on properties. Therefore we need an ontology layer on top of RDF/RDFS for the Semantic Web.

2.3 Web Ontology Language

The Web Ontology Working Group of W3C has identified a number of characteristic use-cases for the Semantic Web that indicate the need for a more powerful ontology modeling language. For machines to perform useful automatic reasoning tasks on Web documents, the language machines must go beyond the expressiveness than RDF and RDF Schema offer. As a result, W3C has defined Web Ontology Language (called *OWL*) [68] as standard for Web ontologies. OWL builds upon RDF and RDF Schema and facilitates greater machine readability of Web content, by providing additional vocabulary along with formal semantics.

Generally an *ontology* describes a domain of discourse. It formally consists of a list of terms and the relationships between them in order to represent an area of knowledge. The terms denote important concepts, such as classes of objects. Besides subclass relationships, ontologies may include information, such as properties, value restrictions, disjoint statements e.t.c. In the context of the Web, ontologies provide a shared understanding of a domain. Such a shared understanding is necessary to overcome differences in terminology. This is a way for a program to know when any two given terms are being used to mean the same thing.

An ontology language permits the development of explicit, formal conceptualizations of models. The main requirements of an ontology language are a well-defined syntax, a formal semantics, convenience of expression, an efficient reasoning support system, and sufficient expressive power.

RDF and RDF Schema allow the representation of some ontological knowledge. The main modeling primitives of RDF and RDFS concern the organization of vocabularies in typed hierarchies. However, a number of other features are missing, such as local scope of properties, disjointness of classes, cardinality of restrictions e.t.c. Therefore we need an ontology language that is richer than RDF Schema, with respect to these additional features. In designing such a language the trade-off is between expressive power and efficient reasoning support. Generally speaking, the richer the language is, the more inefficient the reasoning support becomes. Thus an ontology language in needed that can be supported by reasonably

efficient reasoners while being sufficiently expressive.

The full set of requirements for an ontology language seems unreachable. These requirements have prompted W3C to define OWL to include three different sublanguages (*OWL Full*, *OWL DL*, *OWL Lite*) in order to offer different balances of expressive power and efficient reasoning.

The entire language is called OWL Full and uses all the OWL languages primitives and allows their combination with RDF and RDFS. The advantage of OWL Full is that it is fully compatible with RDF, both syntactically and semantically. The disadvantage of OWL Full is that the language is undecidable, and thus cannot provide efficient reasoning support.

OWL DL is a sublanguage of OWL Full that restricts how the constructors from OWL and RDF may be used. This language corresponds to a well studied description logic [11]. The advantage of OWL DL is that permits efficient reasoning support. The disadvantage is the lose of full compatibility with RDF.

An even further restriction of OWL DL to a subset of the language constructors produces a subset of the language called OWL Lite. The advantage of this language is that it is both easier to understand and easier to implement for tool builders. The disadvantage is of course a more restricted expressiveness.

2.4 Logic and Rules on the Semantic Web

At present, the highest layer that has reached W3C standardization is the web ontology language OWL. The next steps in the development of the Semantic Web will be the realization of more advanced representation and reasoning capabilities for web applications. This is the development of the next layers, logic and proof.

Logic is the study of the principles of reasoning. In general, logic constructs formal languages for expressing knowledge, well-understood formal semantics, and automatic reasoners to deduce (infer) conclusions. It is the foundation of knowledge representation, which has been studied in the area of artificial intelligence, particularly in the form of predicate logic (also known as first-order logic).

Both RDF and OWL (Lite and DL) can be viewed as specializations of predicate logic that are used for Web knowledge representation. This correspondence can be illustrated by describing the semantics of RDF and OWL in the form of logical axioms. They are languages that define reasonable subsets of logic and provide a syntax that fits well with Web languages. As we mentioned before, OWL Lite and OWL DL correspond roughly to a description logic, a subset of predicate logic, for which efficient proof systems exist.

Because of the existence of proof systems, it is possible to trace the proof that leads to a logical consequence. This is an important advantage of logic, that it can provide explanations for answers. Ultimately, since the logic provides traceable steps in obtaining and backtracking a conclusion, an explanation will trace an answer back to a given set of facts and the inference rules used. Explanations are important for the Semantic Web because they establish validated proofs for the

Semantic Web agents in providing reliability for their results.

A key ingredient of logic and proof layers will be *rules*. An inference rule is a scheme for constructing valid inferences. It can be viewed as relations holding between a set of formulas called premises and conclusions, whereby the conclusion is said to be inferable (or derivable or deducible) from the premises. Rule technologies are by now well-established and no longer restricted to AI systems. There seems to be a general consensus that rules with a well-defined semantics are needed in the Semantic Web applications and that they should be well integrated with the ontology level.

Rule systems can be utilized in two stages of the layered development of the Semantic Web:

(a) in the ontology layer, they can serve as extensions of, or alternatives to, description logic-based ontology languages by enriching them with more expressive power and representational capabilities; and

(b) on top of the ontologies, they can be used to develop automated reasoners that can deduce new knowledge based on the ontology knowledge.

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 provide a high-level description, abstracting from implementation details; they are concise and simple to write. They are well-known, understood by non-experts, and well integrated in the mainstream Information Technology.

Therefore integration of rules and ontologies is a subject of active research. The existing proposals for using rules in the Semantic Web refer to rule formalisms originating from the field of Logic Programming and non-monotonic reasoning. These classes of rules are based on different kinds of logics and thus they have well defined declarative semantics, supported by well-developed reasoning algorithms. The simplest language of this kind, playing important role in logic programming, is *Horn logic* or *rule systems*, a subset of predicate logic that allows efficient reasoning. A rule has the form

$$H \longleftarrow A_1, .., A_n$$

where H, A_i are atomic formulas. H is called the *head* or the consequent and $A_1, A_2, ..., A_n$ is called the *body* or *antecedent* of the rule. The rule is read as if $A_1, A_2, ..., A_n$ are known to be true, then *H* is also true.

Horn logic is the basis of monotonic rules. A rule is called monotonic if a conclusion remains valid even if new knowledge becomes available within predicate logic. It appears that the best one can do at present is to take the intersection of the expressive power of Horn logic and description logics.

Among the studies and approaches regarding integration of description logics and rule systems for the Semantic Web, we distinguish:

- Description Logic Programs (*DLP*) [42], which define an intersection of OWL DL and horn clauses, thus making possible re-use of existing reasoners. The resulting logic is decidable, but does not increase the expressive power of OWL, which is the main objective.
- Semantic Web Rule Language (*SWRL*) [47], which extends OWL to include Horn-like rules, but results in an undecidable logic. SWRL is presented in section 2.8.2.
- *CARIN* [53] is defined as a family of languages that provide a hybrid integration of Datalog with different description logics.
- the study made by Rosati [75], dealing with reasoning in description logic knowledge bases augmented with Datalog rules.

In the next section we present several systems which integrate rule languages with ontology languages, following different implementation techniques and allowing different rule language extensions. All these rule-based systems use knowledge representation and inference mechanisms to reach conclusions from facts and rules. The basic distinction lies in the way the inference engines apply the rules to a specific problem. In rule-based systems, there are two directions, forward and backward chaining.

In backward chaining, the system is given a hypothesis or a goal and backtracks to check if there is data available that will support any of these goals. So it searches for rules whose conclusions match this goal. In forward chaining, the system is given data and chains forward, by using the inference rules, in order to reach a goal. When this data is the body of a rule, it triggers this rule and add its head to the conclusions.

2.5 Integrating Rules and Ontologies

2.5.1 CWM

CWM [27] is a forward-chaining first-order inference engine that is written in Python by Tim Berners-Lee and Dan Connolly. It is pronounced as *coom*. It is an open-source program under the W3C software license, used in Semantic Web applications. CWM is a general-purpose processor for text-based data in files for the Semantic Web. Its core language is RDF, which is extended to include rules. CWM supports RDF in three different graph serialization formats: RDF/XML,

N-Triples and Notation 3 [63](*N3*). N3 is a shorthand non-XML serialization format of RDF models, designed due to the lack of human-readability of RDF/XML syntax. It is compact and allows greater expressiveness.

Suppose that in a movie database community, the information of which is represented in RDF/S ontologies, the following rule stands for its members:

If someone is a fan of an actor, then buys a ticket for watching on cinema any new film, where this actor participates in.

This rule can be encoded in CWM as follows:

```
@prefix imdb: <http://www.imdb.com/ontology#> .

{ ?M imdb:isFanOf ?A. ?A imdb:participates ?F. } =>
    { ?M imdb:buyTicket ?F. }.
```

One main feature of CWM is that it can perform several built-in functions like comparing strings, mathematical operations, retrieving resources from the Web (rules or triples), representation capabilities, time handling e.t.c. It also uses a full set of rules in a module, in N3 format, that defines most of the RDFS semantics and can apply these entailment rules in RDF graphs, in order to infer new triples.

2.5.2 Jena

Jena [50] is an open source Semantic Web framework for Java. It is a system that supports many features and integrates RDF/RDFS and OWL with several inference engines. It provides:

- a programming environment for reading and writing RDF, in the formats of RDF/XML, N-Triples and N3, and an OWL API too.
- the RDF Query Language of SPARQL [80]
- apart from the in-memory storage, persistent storage of RDF data in relational databases. Currently are supported the database engines of MySQL, HSQLDB, PostgreSQL, Oracle, Microsoft SQL Server and Apache Derby.

Jena has the functionality to allow a range of different inference engines to be plugged. It provides the following predefined set:

- *RDFS reasoner*
- *OWL reasoner*
- *Transitive reasoner*
- *Generic rule reasoner*

RDFS reasoner supports the use of RDF Schema. It implements almost all of the RDFS entailments through axioms and rules, which are used to derive additional RDF assertions. The RDFS reasoner can work in three different modes: a) *full*, which implements almost all of the RDFS axioms, but it is the most expensive mode, b) *default*, a more restricted mode, c) *simple*, which omits the axioms, but is the most useful mode, according to the authors.

OWL reasoner consists of three implementations: *default*, *mini* and *macro*. Each of the configurations is a sound implementation of a subset of OWL/full semantics but none of them is complete.

Transitive reasoner is the core engine that implements just the transitive and symmetric properties of rdfs:subPropertyOf and rdfs:subClassOf. Although it is a pure inference engine on its own, it is used to build more complex reasoners (e.g. used by RDFS reasoner), in order to provide slightly higher performance, and somewhat more space efficiency.

Generic rule reasoner is a general purpose inference engine, which is also used to implement both the RDFS and OWL reasoners, by supporting rule-based inference over RDF graphs, but it is also available for general tasks. It has the special feature of supporting forward chaining, backward chaining and a hybrid mode of combining them. It provides the forward RETE inference engine and a backward chaining datalog engine that supports tabling. The generic rule reasoner has the option of employing both of the individual rule engines in conjunction, in a hybrid inference engine. Finally this reasoner can also be extended by registering new procedural primitives.

So the rule from the movie database example can be encoded in two ways. Using the backward chaining the rule is:

```
(?M imdb:buyTicket ?F) <-
    (?M imdb:isFanOf ?A), (?A imdb:participates ?F) .
```

The same rule can be encoded using the forward chaining, by exchanging the directions:

```
(?M imdb:isFanOf ?A), (?A imdb:participates ?F) ->
    (?M imdb:buyTicket ?F) .
```

2.5.3 Triple

Triple [78] is a Semantic Web engine that is designed for querying, reasoning and transforming RDF models under several different semantics. Its reasoning engine is based on Horn Logic ad it borrows many basic features from F-Logic, but is specialized for the requirements of the Semantic Web. It supports RDF, RDF Schema and a subset of OWL Lite. It can be represented in a Prolog-like syntax and in an RDF-based, allowing interoperability across the Web.

Horn Logic is the core rule language, but in order to support basic RDF constructs like namespaces, resources and statements, it is syntactically extended.

Triple provides extensions for supporting RDF schema through rules that axiomatize the RDFS semantics (incomplete entailment though). It also provides modules that implement RDF Schema and description logic languages (OWL, DAML+OIL), by interacting with external reasoning components. In the latter case, Triple behaves as hybrid rule language, because it works on top of the ontologies and uses the vocabulary defined in description logic. It finally provided many features, like reified statements, path expressions, skolem functions, modal functionalities in agent communication e.t.c.

Triple rule language can be compiled into logic programs, by using Prolog systems like XSB [91]. Thus Triple is a backward chaining inference engine with tabling support, which guarantees termination of inference.

The rules from the example can be encoded in Triple as:

```
rdf := "http://www.w3.org/1999/02/22-rdf-syntax-ns#".

rdfs :="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#".

@fanticket {
    imdb := "http://www.imdb.com/ontology#".
    FORALL A,B  A[imdb:buyTicket->B] <-
        EXISTS C    ( A[imdb:isFanOf->C] AND
                      C[imdb:participates->B]).
}
```

2.5.4 SWI-Prolog Semantic Web Library

Swi-Prolog [82] is an open-source Prolog system which is widely used in research and education as well as for commercial applications. It has a rich set of features, by providing many add-ons and libraries. One of these additional functionalities is provided by the SWI-Prolog Semantic Web Library, which deal with RDF/S documents. It consists of Prolog packages for reading, querying and storing RDF triples, only in RDF/XML format. It is a hybrid rule system with backward inference engine, because the predicates of the Prolog rules and the ontology are distinguished and suitable interfacing between them is facilitated. If a user wants to explore the hierarchies and domain restrictions of RDFS constructs, he refers to the special predicates, provided by the Semantic Web library.

Therefore, using this library, the rule from the example is written in Prolog as:

```
'http://www.imdb.com/ontology#buyTicket' (Member,Film) :-
    rdf(Member,'http://www.imdb.com/ontology#isFanOf',Actor),
    rdf(Actor,'http://www.imdb.com/ontology#participates',Film).
```

2.5.5 Characteristics of Rule-based Systems

In this section we presented systems that support reasoning with RDF/S ontologies. As we stated before, all these systems use as rule formalism a logic programming language, which is based on definite Horn clauses, with sound and complete proof

procedures. CWM, Jena, TRIPLE are homogeneous reasoners, which means that both ontologies and rules are embedded in a logical language, without making a priori distinction between the rule predicates and the ontology predicates. These systems use modules that contain the axiom schemes for the several forms of entailment defined for RDF graphs. On the other hand, as we said in subsection 2.5.4, SWI-Prolog Library can be seen as a hybrid reasoner. Furthermore, CWM is a forward engine while TRIPLE and SWI-Prolog are backward ones. Jena supports both forward and backward chaining, and even can combine them together. Triple relies on XSB-Prolog tabling features for guaranteeing termination of inference, while Jena also has its own implementation of tabling, inspired by the mechanisms of XSB-Prolog too.

2.6 Nonmonotonic Rules on the Semantic Web

One of the issues that have attracted the concentration of the developers of the Semantic Web, is the nature of the rule systems that should be employed in the layered development of the Semantic Web. Most studies have focused on the employment of monotonic logical systems, because the Semantic Web standards (RDF, RDFS, OWL) are based on classical predicate logic. Although important and useful in many situations, like in factual and ontological knowledge, which contains general truths that do not change often, approaches based on monotonic reasoning suffer from not dealing with inconsistent information properly. When an inconsistency arises in a knowledge base, then every conclusion can be derived.

Nonmonotonic rule systems seem also to be a good solution, as they offer more expressive capabilities and are closer to commonsense reasoning. There are many scenarios in which conflicting rules may arise on the Web. Here we mention a few of them:

- *Reasoning with Incomplete Information:* [1] describes a scenario where business rules have to deal with incomplete information: in the absence of certain information some assumptions have to be made which lead to conclusions not supported by classical predicate logic. In many applications on the Web such assumptions must be made because other players may not be able (e.g. due to communication problems) or willing (e.g. because of privacy or security concerns) to provide information. This is the classical case for the use of nonmonotonic knowledge representation and reasoning [60].

- *Rules with Exceptions:* Rules with exceptions are a natural representation for policies and business rules [7], and priority information is often implicitly or explicitly available to resolve conflicts among rules. Potential applications include security policies [10], [54] business rules [1], personalization, brokering, bargaining, and automated agent negotiations [36].

- *Default Inheritance in Ontologies:* Default inheritance is a well-known feature of certain knowledge representation formalisms. Thus, it may play a

role in ontology languages, which currently do not support this feature. [45] presents some ideas for possible uses of default inheritance in ontologies. A natural way of representing default inheritance is rules with exceptions, plus priority information. Thus, nonmonotonic rule systems can be utilized in ontology languages.

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

Nonmonotonic reasoning is a subfield of Artificial Intelligence trying to find more realistic formal models of reasoning than classical (first-order) logic. A logic is monotonic if the truth of a proposition does not change when new information are added to the system. By contrast, the real world requires common sense reasoning, that deals with incomplete and potentially inconsistent information and one draws conclusions that have to be withdrawn when further information is obtained. In a nonmonotonic logic, the set of conclusions, in contrast to monotonic logic, does not grow monotonically (in fact, it can decrease) with the given information. This is the phenomenon that nonmonotonic reasoning methods try to formalize. Several nonmonotonic logics have been proposed and studied during the last few several decades, among them default logic, autoepistemic logic, circumscription and defeasible logic.

Default logic [73] is a non-monotonic logic proposed by Raymond Reiter and it has been used to formalize a number of different reasoning tasks. Default Logic assumes that knowledge is represented in terms of a default theory. It can express facts like "by default, something is true". A default theory is a pair (D,W). W is a set of first order formulas, called the background theory, representing the facts that are known for sure. D is a set of default rules of the form

$$\frac{A:B_1,..,B_N}{C}$$

where A, B_i, C are classical closed formulas. This has the intuitive reading: if *A* is provable and $\forall i \in [1, N] \Longrightarrow \neg B_i$ is not provable, then derive *C*. *A* is called the prerequisite, B_i a consistency condition or justification, and *C* the consequent of the default. We can make this clear by formalizing the default rule as

$$\frac{bird(X):flies(X)}{flies(X)}$$

in combination with the rule

$$penguin(X) \longrightarrow \neg flies(X)$$

According to this rule , if *X* is a bird, and it can be assumed that it flies, then we can conclude that it flies. One of the exceptions to this rule is the penguin.

Moore 's autoepistemic logic [62] is the most widely studied logic of a class called modal nonmonotonic logics. The autoepistemic logic is a formal logic aimed at formalizing representation and reasoning of knowledge about knowledge. It can express knowledge and lack of knowledge about facts. These logics use a modal operator to express explicitly that a certain formula is consistent or believed. Moore extends the syntax of propositional logic by a modal operator *L* indicating knowledge: if *p* is a formula then also *Lp* is. *Lp* stands for "*p* is believed". The idea is to allow reasoning about what an agent completely knows and about what he does not know. This means that: if *p* belongs to the set of beliefs *B*, then also *Lp* has to belong to *B*, otherwise ¬ *Lp* must be in *B*.

Circumscription [61] was created by McCarthy and it has generated a great deal of interest in the nonmonotonic reasoning community. It formalizes the common sense assumption that things are as expected unless otherwise specified. Circumscription deals with the minimization of predicates subject to restrictions expressed by predicate formulas. In circumscription, theories are written in classical first-order logic, however the entailment relation is not classical.

Defeasible logic [65], [5] is a nonmonotonic logic proposed by Donald Nute to formalize defeasible reasoning. Defeasible reasoning is a simple, but often more efficient than other nonmonotonic rule systems rule-based approach, to reasoning with incomplete and inconsistent information. It is based on the use of rules that may be defeated by other rules. In general, a knowledge base in defeasible logic consists of five different kinds of knowledge: facts, strict rules, defeasible rules, defeaters, and a superiority relation among rules.

Defeasible logic has recently been used in Semantic Web applications. In particular, there are implementations of defeasible logic that interoperate with Semantic Web standards. We will present these implementations in the next section. Its use as a rule language for Semantic Web applications has many advantages. It offers enhanced representational capabilities allowing one to reason with incomplete and contradictory information. It also can reason both with static and dynamic knowledge on the Semantic Web. Defeasible logic also supports, in contrast to the classical nonmonotonic reasoning approaches, the representational feature of priority. Priorities on rules may be used to resolve some conflicts among them. Finally, compared to mainstream nonmonotonic reasoning, defeasible logic has the additional very important advantage of its relatively low computational complexity [56].

2.7 Nonmonotonic Rule Systems on the Semantic Web

We present the following implemented systems, which follow nonmonotonic reasoning approaches for Semantic Web applications:

2.7.1 DR-Prolog

DR-Prolog [2] is a defeasible reasoning system on the Web, in which is based the implementation of our system. It is a powerful rule system based on defeasible logic, which combines the expressive power of a nonmonotonic logic with the major Semantic Web standards (RDF/S, OWL, and RuleML), to build applications for the logic and proof layers of the Semantic Web. DR-Prolog has a firm formal foundation provided by a number of papers published in top artificial intelligence and logic programming conferences and journals [5], [9], [4], [57], [56], [58], [59]. The main characteristics of DR-Prolog are the following:

- It is based on Prolog. The core of the system consists of a well-studied translation [6] of defeasible knowledge into logic programs under Well- Founded Semantics [32]. This declarative translation distinguishes from other defeasible reasoning systems.
- The main focus is on flexibility. Monotonic and nonmonotonic rules, preferences among rules are part of the interface and the implementation. It also supports open and closed world assumption, and reasoning with inconsistencies. DR-Prolog implements the entire of defeasible logic and supports a number of reasoning variants.
- The system can reason with RDF data and RDF Schema and (parts of) OWL ontologies. The latter happens through the transformation of the RDFS constructs and many OWL constructs into rules. Note however, that a number of OWL constructs cannot be captured by the expressive power of rule languages.
- Its user interface is syntactically compatible with RuleML, the main standardization effort for rules on the Semantic Web.

2.7.2 DR-DEVICE

DR-DEVICE [12] is also a defeasible reasoning system for the Semantic Web, exhibiting similar functionality with DR-Prolog, but following a different overall approach. It is a powerful query answering system that is implemented in Jess. It is capable of reasoning about RDF data and RDF Schema ontologies over the Web using defeasible logic. DR-DEVICE implements the full version of defeasible logic, by supporting multiple rule types of defeasible logic, priorities among rules, two types of negation, multiple variants e.t.c. Its architecture is based on the CLIPS rule system, and is in fact an extension of R-Device: a system for rules on RDF

data. It is syntactically compatible with RuleML and the rules can be expressed either in the rule language of CLIPS, or in an extension of the OO-RuleML syntax.

DR-DEVICE is based on a translation of defeasible theories into the non-logical language of Jess, with an associated loss in declarativity of the overall approach. On the other hand, it has advantages of easier integration with mainstream software technologies.

2.7.3 SweetJess

SweetJess [45] is another defeasible reasoning system that it is implemented in Jess and integrates well with RuleML. It is based on Situated Courteous Logic Programs (*SCLP*), a powerful knowledge representation formalism that supports prioritized conflict handling and procedural attachments. The latter is a feature not supported by any of the other nonmonotonic implementations. It uses Jess rule engine as inference engine and supports translation from rules in SCLP, syntactically encoded in RuleML, into Jess rules and vice versa. An integration effort with the Semantic Web ontology language of DAML+OIL exists, in which SCLP RuleML is extended and is called *DamlRuleML*.

SweetJess is more limited in flexibility, in that it implements only one reasoning variant (ambiguity blocking variant). Moreover, it imposes a number of restrictions on the programs it can map on Jess.

2.7.4 dlvhex

dlvhex [29] is a Semantic Web engine that is based on *HEX-programs*, which are an extension of Answer-Set Programs. Answer-set programming (*ASP*) is a declarative programming approach, similar in syntax to logic programming, with nonmonotonic semantics. HEX-programs are high-order logic programs, with external atoms for software interoperability, which extend answer-set semantics. It is the only nonmonotonic rule system that supports high-order features, which are only supported by TRIPLE, from the monotonic systems we presented before. dlvhex integrates rules on top of ontologies, by dealing with external knowledge, through external atoms. External atoms allow integration of external sources of knowledge, like RDF data, and reasoners of various nature, like description-logic reasoners for OWL DL. In contrast to the previous homogeneous nonmonotonic systems, dlvhex is a hybrid approach.

2.8 Rule Languages for the Semantic Web

Most of the rule-based systems, that we presented in the previous section and have been developed over the time, present different concepts of rule languages and notations to feed the rules into the systems. Rule systems use languages that are best suited for their intentions and capabilities. A different research effort deal with a unifying framework to represent rules for the Semantic Web context, where

rules need to be published on the Web and implemented in such a way, as to allow software agents to process on them. A standard XML encoding for the rules is needed that would ease the exchange of rules on the Web between agents. We present two standardization efforts in the area of rules for the Semantic Web:

2.8.1 Rule Markup Language

The Rule Markup Language (*RuleML*) [76] is a XML based rule language standard that was developed to express both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks on the Web. It is developed by the Rule Markup Initiative, an open network of researchers and practitioners from several countries that was formed to develop a canonical Web language for rules using XML markup, formal semantics, and efficient implementations.

RuleML provides a classification of the rule it supports. RuleML contains a hierarchy of rules, including reaction rules (event-condition-action rules), transformation rules (functional-equational rules), derivation rules (implicational-inference rules), also specialized to facts (premiseless derivation rules) and queries (conclusionless derivation rules), as well as integrity-constraints (consistency-maintenance rules). For these top-level families, XML DTDs and Schemas are provided, reflecting the structures of the rule families. Derivation rules, facts, and queries have been developed mostly up to date.

The RuleML hierarchy of rules branches into the two direct categories of reaction rules and transformation rules. On the next level, transformation rules specialize to the subcategory of derivation rules. Then, derivation rules have further subcategories, namely facts and queries. Finally, queries specialize to integrity constraints. More subdivisions are being worked out, especially for reaction rules. A graphical view of the top-level classification of RuleML rules is shown in Figure 2.5.

The basic subcategory in this hierarchy is the language of function-free Horn clauses, known as Datalog. This is a sublanguage of derivation-rules and Horn logic and it is the foundation for the kernel of RuleML sublanguages. Datalog is the language in the intersection of SQL and Prolog. Its syntax is defined by an XML Schema, using XML tags such as <head>, <body>, <atom> and is referred as RuleML proposal. The latest XSD version that has been released is RuleML version 0.91. To explain the Datalog features, we show the following example, where a rule is formalized in RuleML Datalog:

```
<Implies>
  <head>
    <Atom>
      <Rel>buyTicket</Rel>
      <Var>Member</Var>
      <Var>Film</Var>
    </Atom>
  </head>
```

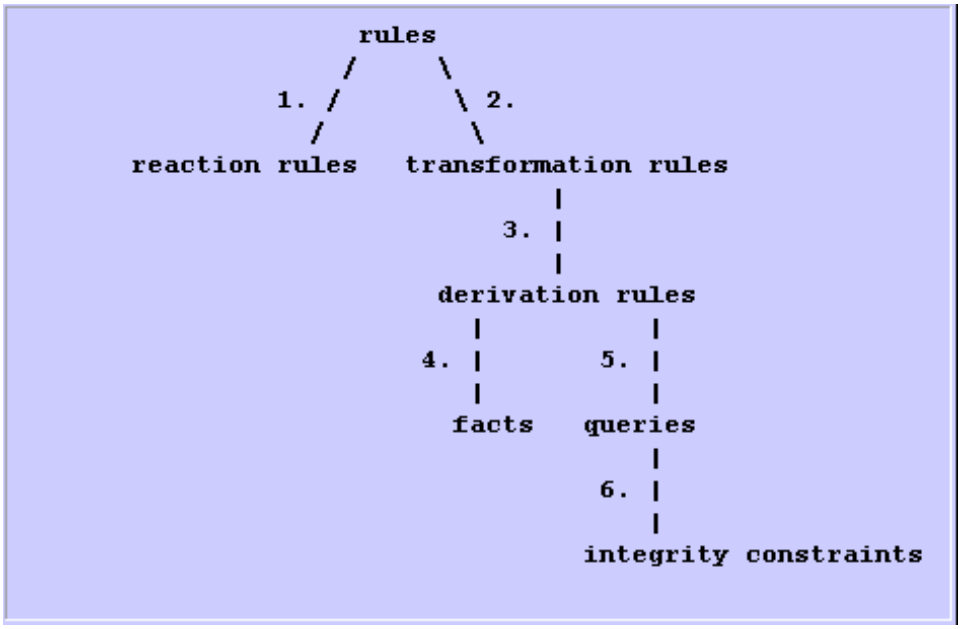

Figure 2.5: The RuleML hierarchy of rules.

```
  <body>
    <And>
      <Atom>
        <Rel>isFanOf</Rel>
        <Var>Member</Var>
        <Var>Actor</Var>
      </Atom>
      <Atom>
        <Rel>participates</Rel>
        <Var>Actor</Var>
        <Var>Film</Var>
      </Atom>
    </And>
  </body>
</Implies>
```

One of the goals of RuleML is to integrate with ontology languages and subsequently with OWL. The current outcome of these efforts is a draft for SWRL, which is based on a combination of the OWL DL with the unary and binary Datalog sublanguages of RuleML. SWRL is described in the next section.

2.8.2 Semantic Web Rule Language

SWRL (Semantic Web Rule Language) [48] is a proposal for a Semantic Web rule language, combining sublanguages of the OWL Web Ontology Language (OWL

DL and Lite) with those of the Rule Markup Language (Unary/Binary Datalog). The language makes it possible to extend the set of OWL axioms to include Horn-like rules. It thus enables Horn-like rules to be combined with an OWL knowledge base.

Rules are of the form of an implication between an antecedent (body) and a consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. Both the antecedent and the consequent consist of zero or more atoms. An empty antecedent is treated as trivially true and an empty consequent is treated as trivially false. Multiple atoms are treated as a conjunction.

The integration between rules and ontologies is achieved by using the concepts and the roles of OWL DL, (which denote, respectively, unary and binary predicates) for building atoms of the SWRL rules. The concepts and the roles are defined by axioms expressed in OWL, which is a subset of SWRL, and used in SWRL rules. Atoms in these rules can be of the form C(x), P(x,y), sameAs(x,y) or differentFrom(x,y), where C is an OWL description, P is an OWL property, and x,y are either variables, OWL individuals or OWL data values.

SWRL provides an XML Concrete Syntax which is a combination of the OWL XML and RuleML presentation syntax. An example is the following:

```
<ruleml:imp>
  <ruleml:_body>
    <swrlx:individualPropertyAtom  swrlx:property="buyTicket">
            <ruleml:var>member</ruleml:var>
            <ruleml:var>film</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom  swrlx:property="isFanOf">
            <ruleml:var>member</ruleml:var>
            <ruleml:var>actor</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:individualPropertyAtom  swrlx:property="participates">
      <ruleml:var>actor</ruleml:var>
      <ruleml:var>film</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_head>
</ruleml:imp>
```

Chapter 3

Extension of Defeasible Logic

3.1 Defeasible Logic

Defeasible reasoning is a nonmonotonic reasoning approach in which the gaps due to incomplete information are closed through the use of defeasible rules that are usually appropriate. This reasoning family is comprised of defeasible logics [65], [5] and Courteous Logic Programs [43]. Defeasible logics were introduced and developed by Nute [65] over several years. These logics perform defeasible reasoning, where a conclusion supported by a rule might be overturned by the effect of another rule. These logics also have a monotonic reasoning component, and a priority on rules. One advantage of Nute 's design was that it was aimed at supporting efficient reasoning and he kept the language as simple as possible.

Defeasible logic has recently attracted considerable interest. Its use in various application domains has been advocated, including the modelling of regulations and business rules [3], modelling of contracts [35], legal reasoning [41], agent negotiations [36], [79], modelling of agents and agent societies [40], [38], and applications to the Semantic Web [2], [12].

Being nonmonotonic, defeasible logic deal with potential conflicts (inconsistencies) among knowledge items. Thus, it contain classical negation, contrary to usual logic programming systems. It can also deal with negation as failure (NAF), the other type of negation typical of nonmonotonic logic programming systems; in fact, Wagner [88] argues that the Semantic Web requires both types of negation. In defeasible logic, it is often assumed that NAF is not included in the object language. However, as Antoniou et al. [9] show, it can be easily simulated when necessary. Thus, we may use NAF in the object language and transform the original knowledge to logical rules without NAF exhibiting the same behavior.

The logics take a pragmatic view and have low computational complexity. This is, among other things, achieved through the absence of disjunction and the local nature of priorities: Only priorities between conflicting rules are used, as opposed to systems of formal argumentation where more complex kinds of priorities (e.g., comparing the strength of reasoning chains) are often incorporated.

Defeasible logic is a logical formalism that has been studied and analyzed, with strong results in terms of proof theory [5], semantics [37], [57] and computational complexity [56]. As a consequence, its translation into logic programs, a fundamental part of our implemented system, has also been studied thoroughly [59], [8],[6].

3.1.1 Syntax

A defeasible theory (a knowledge base in defeasible logic) 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.

Facts are literals that are treated as known knowledge (given or observed facts of a case), for example, Tweety is an emu. Written formally, this would be expressed as

$$emu(tweety).$$

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 "*Emus are birds*". Written formally:

$$emu(X) \rightarrow bird(X).$$

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 "Birds typically fly"; written formally:

$$bird(X) \Rightarrow flies(X).$$

The idea is that if we know that something is a bird, then we may conclude that it flies, *unless there is other, not inferior, evidence suggesting that it may not fly*.

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. In other words, we do not wish to conclude $\neg flies(X)$ if *heavy(X)*; we simply want to prevent a conclusion *flies(X)*.

The *superiority relation* among rules is used to define priorities among rules, i.e., where one rule may override the conclusion of another rule. For example, given the defeasible rules

$$r : bird(X) \Rightarrow flies(X)$$

$$s : brokenWing(X) \Rightarrow \neg flies(X)$$

which contradict one another, no conclusive decision can be made about whether a bird with broken wings can fly. But if we introduce a superiority relation $>$ with s $>$ r, with the intended meaning that s is strictly stronger than r, then we can indeed conclude that the bird cannot fly.

Notice that a cycle in the superiority relation is counterintuitive. In the above example, it makes no sense to have both $r > s$ and $s > r$. Consequently, we focus on cases where the superiority relation is acyclic.

Another point worth noting is that, in defeasible logic, priorities are local in the following sense: two rules are considered to be competing with one another only if they have complementary heads. Thus, since the superiority relation is used to resolve conflicts among competing rules, it is only used to compare rules with complementary heads; the information r $>$ s for rules r, s without complementary heads may be part of the superiority relation, but has no effect on the proof theory.

3.1.2 Formal Definition

In this report we restrict attention to essentially propositional defeasible logic. Rules with free variables are interpreted as rule schemas, that is, as the set of all ground instances. If q is a literal, $\sim$q denotes the complementary literal (if q is a positive literal p then $\sim$q is $\neg$p; and if q is $\neg$p, then $\sim$q is p).

Rules are defined over a *language* (or *signature*) Σ, the set of propositions (atoms) and labels that may be used in the rule.

A rule r: *A(r)* $\hookrightarrow$ *C(r)* consists of its unique *label r*, its *antecedent A(r)*, (*A(r)* may be omitted if it is the empty set) which is a finite set of literals, an arrow $\hookrightarrow$ (which is a placeholder for concrete arrows to be introduced in a moment), and its *head* (or *consequent*) *C(r)* which is a literal. In writing rules we omit set notation for antecedents, and sometimes we omit the label when it is not relevant for the context. There are three kinds of rules, each represented by a different arrow. Strict rules use $\rightarrow$, defeasible rules use $\Rightarrow$, and defeaters use $\rightsquigarrow$.

Given a set *R* of rules, we denote the set of all strict rules in *R* by R_s, the set of strict and defeasible rules in *R* by R_{sd}, the set of defeasible rules in *R* by R_d, and the set of defeaters in *R* by R_{dft}. *R[q]* denotes the set of rules in *R* with consequent q.

A *superiority relation on R* is a relation $>$ on *R*. When $r_1 > r_2$, then r_1 is called *superior* to r_2, and r_2 *inferior* to r_1. Intuitively, $r_1 > r_2$ expresses that r_1 overrules

r_2, should both rules be applicable. Typically we assume $>$ to be acyclic (that is, the transitive closure of $>$ is irreflexive).

A *defeasible theory* D is a triple (F, R, $>$) where F is a finite set of literals (called *facts*), R a finite set of rules, and $>$ an acyclic superiority relation on R. D is called *decisive* if the atom dependency graph of D is acyclic.

3.1.3 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),...,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
$\exists r \in R_s[q]\ \forall a \in A(r)$: $+\Delta a\ \epsilon\ P$(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).

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

$\forall r \in R_s[q]\ \exists a \in A(r)$: $-\Delta a \in P(1..i)$

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 -Δp.

$+\partial$: 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 condition -∂. It is nothing more than a strong negation of the condition +∂.

$-\partial$: 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 $t \not> s$

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 [43]. 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).

Proposition 1. [14] *If D is decisive, then for each literal p:*
(a) either $D \vdash +\Delta p$ *or* $D \vdash -\Delta p$
(a) either $D \vdash -\partial p$ *or* $D \vdash +\partial p$

Not every defeasible theory satisfies this property. For example, in the theory consisting of the single rule $p \Rightarrow p$ neither $-\partial p$ nor $+\partial p$ is provable. The proof of the proposition can be found in [14].

Proposition 2. [5]*Consider a defeasible theory D.*
(1) *If* $D \forall -\Delta \sim p$ *and* $D \forall +\Delta p$ *then* $D \forall +\partial p$.
(2) *If* $D \vdash +\Delta \sim p$ *and* $D \vdash -\Delta p$ *then* $D \vdash -\partial p$.
(3) *If* $D \vdash +\partial \sim p$ *and* $D \vdash -\Delta p$ *and* $D \forall -\partial p$ *then D is cyclic*.

Theorem 3. [5]*If D is an acyclic defeasible theory, then D is conclusion equivalent to a theory D' that contains no use of the superiority relation, nor defeaters. If D is a cyclic defeasible theory, then D is conclusion equivalent to a theory D' that contains no use of defeaters, and if D' contains cycles then they have length 2, and each cycle involves the only two rules for a literal and its complement.*

Proposition 4. [5]*Let D be an acyclic defeasible theory.*
If $D \vdash +\partial p$ *and* $D \vdash +\partial \sim p$ *then* $D \vdash +\Delta p$ *and* $D \vdash +\Delta \sim p$.

Consequently, if D contains no strict rules and no facts and $D \vdash +\partial q$ *, then* $D \vdash -\partial \sim q$.
The proves for the Propositions 2, 4 and for the Theorem 3 can be found in [5].

Governatori *et. al* [37] describe defeasible logic and its variants in argumentation-theoretic terms. Under the argumentation semantics, proof trees are grouped together as arguments, and conflicting arguments are resolved by notions of argument defeat that reflect defeat in defeasible logic.

Maher [55] gives a denotational-style semantics to defeasible logic, providing another useful analysis of this logic. The semantics is compositional, and fully abstract in all but one syntactic class.

A model-theoretic semantics semantics of defeasible logic is given by Maher in [57]. This semantics follows Nute's semantics for LDR [64] in that models represent a state of mind or "belief state" in which definite knowledge (that which is "known") is distinguished from defeasible knowledge (that which is "believed"). A major difference from [64] is that adopts partial models as the basic from which to work.

3.2 Modelling Agents

As we stated, one of the basic motivations of our work is the modelling of multi-agent systems. At this point, we should present a definition of the term *agent*. Although there is no universally accepted definition of this term, we present the definition, adapted from [89] : "*An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives*".

As the definition said, we consider agents to be systems that are situated or embodied in some environment. By this, we mean that agents are capable of sensing their environment and they have a set of possible actions that they can perform, in order to modify their environment. Agents cannot perform all actions in all situations. Therefor actions have preconditions, which define the possible situations in which they can be applied. In almost all realistic applications, agents have at best a partial control over their environment, in that it can influence it, but not a complete control.

There is a wide range of environments with different properties, that can be occupied by agents. Such a case is when an agent is situated in part of the real world, where it senses its environment by physical sensors and actions are physical, like moving objects around. There are also many cases where software agents perform actions in a software environment. An example is the *buyer agents* or shopping bots that help Internet users find products and services they are searching for. These software applications recommend products that were visited by users who did the same search. An other example are *user agents*, which execute tasks automatically for the user, like sorting emails according to the user's order of preference. Other Web applications of software agents include spam filters, search engine bots etc. Finally most *daemons*, background processes which monitor a software environment, in Unix-like systems can be viewed as agents.

3.2.1 Intelligent Agents

An intelligent agent is one that operates flexibly and rationally in a variety of environmental circumstances, in order to meet its design objectives, given the informa-

tion they have and their perceptual and effectual capabilities. These agents should have the following properties:

- *autonomy* : To some extent they have control over their behavior and make independent decisions without driven by humans and other systems;
- *reactivity* : They are able to perceive their environment, and are responsive in a timely fashion to changes that occur in it;
- *pro-activeness* : They are able to exhibit goal-directed behavior by taking the initiative in order to achieve their goals;
- *social ability* : They are capable of negotiating and cooperating with other agents.

One of the key problems in facing an agent is the decision of which actions to perform from the set of possible actions. *Agent architectures* are particular methodologies for building agents for decision making systems that are embedded in an environment. We consider four types of agent architecture, characterized by the different nature of their decision making:

- *Logic-Based Architectures* : The "traditional" approach to building artificially intelligent systems, in which decision making is achieved through logical deduction;
- *Reactive Architectures* : Such systems are perceived as simply reacting to an environment, without reasoning about it. Decision making is implemented via simple direct mapping from situation to action;
- *Belief-Desire-Intention Architectures* : BDI architectures have their roots in the philosophical tradition of understanding practical reasoning, which is the process of deciding which action to perform in furtherance of our goals. Decision making depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent;
- *Layered Architectures* : A class of architectures, where agents are capable of reactive and proactive behavior and various subsystems are arranged into a hierarchy of interacting layers. decision making is realized via the interaction of a number of task accomplishing layers.

3.2.2 Cognitive Agents

A *cognitive* or *rational* agent is an intelligent agent that chooses to perform actions that are in its own best interests, given his or her knowledge of its environment. It behaves in a way that maximizes its chances for goal achievement. Suppose that a cognitive agent has the goal to stay dry and it has the belief that it is raining. Then it chooses to take an umbrella when it leaves its house, since this action is in

its own best interest, which is the goal of staying dry. In our work, we take into account the BDI architecture as the model for building cognitive agents. This is an approach of building Intentional Agent systems, where agents are provided with mental attitudes, like believing, wanting, hoping and fearing and so on.

3.2.3 BDI Architecture

Belief-Desire-Intention (BDI) [71],[72] model of rational agency contains explicit representations of beliefs, desires, and intentions.

Beliefs are the information an agent has about the world. We use the term belief, rather than *knowledge*, to recognize that what an agent believes may be incomplete or incorrect. *Desires* are those things that the agent would like to accomplish or bring about. In BDI architecture we use the term *goals* to represent desires and it adds the further restriction to be consistent with one another. *Intentions* are those desires that an agent has chosen to achieve. An agent is not able to achieve all its desires and these chosen desires, to which it has some commitment, are intentions.

The BDI model has its roots in the philosophical tradition of understanding *practical reasoning* in humans, originally developed by Michael Bratman [17]. Practical reasoning is the process of deciding, moment by moment, what to do. Contrast to theoretical reasoning, which is directed towards beliefs, practical reasoning is the application of reasoning towards action, e.g. the decision to catch a bus instead of a train.

Practical reasoning appears to consist of at least two distinct activities: deciding what goals we want to achieve, and how we want to achieve these goals. The former process is known as *deliberation* and the latter as *means-ends* reasoning. For example, deciding which career to aim for, after graduating from university, is deliberation. The process of deciding a plan, in order to achieve the chosen goal of pursuing a career as an academic, is means-ends reasoning.

Thus intentions play a crucial role in practical reasoning and their most obvious property is that they are *pro-attitudes*, which means that they tend to lead to action, e.g. applying to various PhD programs in order to achieve the intention to become academic. A agent should also at times reconsider some intentions, because it comes to believe that either has achieved them or its current intentions are no longer possible. The relationships that exist between an agent 's mental states, lead it to select and perform rational actions. In this way, it is realized the definition of an agent that we presented in section 3.2. A BDI agent senses its environment through mental states and then chooses to perform actions.

A family of logics that support a formal theory of BDI model has been developed. The first logical description was Anand Rao and Michael Georgeff's *BDICTL*. More recently, BDICTL was extended by Michael Wooldridge to the Logic Of Rational Agents (*LORA*) [90]. This a multi-modal logic, where modalities represent BDI components, combined with temporal and action components.

BDI architectures have been implemented several times. The first implementation was the Intelligent Resource-Bounded Machine Architecture (*IRMA*) [18].

However, the best-known implementation is the Procedural Reasoning System (*PRS*) [33], that was developed by a team led by Georgeff. Many descendants of PRS were implemented later.

3.2.4 Multiagent Systems

Much of traditional Artificial Intelligence has been concerned with the characteristics and the structure of a single agent in order to function intelligently. But intelligent agents do not function in isolation, but they operate and exist in some environment, which typically is both computational and physical and it might contain other agents. Thus an agent cannot operate usefully by itself, but it also interacts with the other agents. These environments, in which agents can operate effectively and interact with each other productively, are called *multiagent systems*.

In [51] are identified the major characteristics of multiagent systems. Multiagent systems provide an infrastructure specifying communication and interaction protocols. Their environments are typically open and decentralized. The agents are autonomous, making decision without regard of the others, and distributed. Difference may exist in agent 's behavior, as they may be self-interested or cooperative. Each agent also has incomplete information about its environment and it is restricted to its capabilities.

One of the most important concerns of a multiagent environment are the communication and interaction protocols. Communication protocols enable the exchange of information among agents, based on a shared system of signs. Interaction protocols enable agents to have conversations through a structured exchange of messages, leading to some defined outcome.

One of the key functionalities needed to implement a multiagent system is the *coordination* as a form of interaction, which is particularly important with respect to goal attainment and task completion. Coordination is the property of system of agents performing actions that fit well with each other, as well as to the process of achieving this state. The purpose is to achieve states of affairs that are considered as desirable by one or several agents and that results in a coherent system, which behaves well as a unit. A measure of coordination is the degree in which activities like livelock, deadlock and increased resource contention are avoided.

In order to coordinate the agents, we must take dependencies among their activities into consideration. Two contrasting manifestations of coordination that play important roles are *cooperation* and *competition*. Cooperation is coordination among nonantagonistic agents that work together, in order to achieve a common goal, and so they succeed or fail together. On the other hand, in the case of competition, several agents with conflicting goals work against each other and so the success of one implies the failure of others. The coordination among competitive agents is called *negotiation*.

3.2.5 Society of Agents and Norms

An intelligent system does not function in isolation but they are part of an environment, which it may contain other such intelligent agent systems. It makes sense to view such systems as a *society* of agents.

A group of agents can form a small society in which they play different roles. When an agent joins a group, he joins in one or more roles, and acquires the commitments of that role. Though it is an autonomous agent, it is constrained by the commitments for the roles it adopts. Thus typically roles include permissions and obligations, and are associated with specific behavioral patterns.

The mental attitudes that describe an agent 's internal state, as in the Belief-Desire-Intention agent that we described in subsection 3.2.3, are appropriate for a number of applications and situations but not for understanding all aspects of social interactions. A self-interested agent need not be selfish and it may have other interests than its immediate personal gain. Thus intelligent agents should interact and coordinate to achieve their own goals and the goals of their society. We use *social laws* and *norms* as a way of coordinating activities and behaviors of large numbers of agents in a society.

Norms are established expected patterns of behavior. They are rules that specify how an agent embedded in a society of agents should behave, striking a balance between individual freedom on the one hand, and the goal of the agent society on the other. They consist a set of constraints on individual actions in particular contexts such that, if all agents follow the social laws, the agent system will avoid undesirable states. As we said, norms carry the meaning of expected behavior, e.g it is a norm to form a queue when waiting for a bus and to allow those who arrived first to enter the bus first.

3.2.6 The BOID Architecture

More recent works on cognitive agents try to combine two different perspectives. The first one is the classical agent systems based on mental attitudes, like the BDI architecture. The other is the artificial societies of agents, where normative aspects coordinate the behaviors of intelligent autonomous agents [23], [31], [69], [26], [39]. The result of this combination of perspectives is the modelling of autonomous agents based on cognitive and social models, where an agent deliberation and behavior is determined as an interplay between mental attitudes and normative aspects. BOID architecture is such an approach, in which BDI cognitive states interact with one another and with obligations as well, incorporating in this way norms and commitments.

Beliefs-Obligations-Intentions-Desires (*BOID*) [23], [24], [25], [28] architecture is an agent architecture that contains at least four components representing the beliefs (B), obligations (O), intentions (I) and desires (D) of the agent. A logical framework has been developed, where the content of each component is represented by sets of propositional logical formulas, often in the form of defeasible

rules. BOID identifies two general types of conflicts: *Internal conflicts* can arise within each component and *external conflicts* between them. These two general types of conflicts can be distinguished into further conflict subtypes, e.g. unary subtypes within each component, binary conflict subtypes, such as OI, BD, e.t.c., ternary conflicts subtypes, such as BID, BOD, e.t.c., and one quadruplicate conflict type BOID. A classification of conflict resolution types among the motivational attitudes is supported, which corresponds to what in agent theories is called *agent types*. For example, an agent is *realistic* if beliefs overrule all the other components, *simple-minded* if intentions overrule desires and obligation, *social* if obligations overrule desires e.t.c.

3.3 Modal Logic

In formal logic, a modal logic is any logic for handling modalities. A modal is an expression that is used to qualify the truth of a judgement. Modal logic is, strictly speaking, the study of the deductive behavior of the expressions "it is necessary that" and "it is possible that". In its general form, modal logic was used by philosophers to investigate different modes of truth, but it also has important applications in computer science. For example, the following are all modal propositions:

It is possible that it will rain tomorrow.
It is necessary that either it is raining here now or it is not raining here now.

However, the term "modal logic" may be used more broadly for handling a number of other ideas. These include logics for belief, for tense and other temporal expressions, for the deontic expressions such as "it is obligatory that" and "it is permitted that", and many others. The best known family of modal logics with similar rules and a variety of different symbols will be described in the next sections.

Modal logic was first developed to deal with the concepts of necessity and possibility, which are called basic modal operators, and only afterward was extended to others. These operators are usually written as $\Box$ for *Necessarily* and $\Diamond$ for *Possibly*.

Many systems of modal logic, with widely varying properties, have been proposed since C. I. Lewis began working in the area in 1910. The most familiar logics in the modal family are constructed from the weakest modal logic, named K in honor of Saul Kripke. The symbols of K include "$\sim$" for "not", "$\longrightarrow$" for "if...then", and the modal operators of "$\Box$" and "$\Diamond$". These operators is definable in terms of the other, forming a dual pair of operators:

- $\Box p$ (necessarily p) is equivalent to $\sim\Diamond\sim p$ (not possible "not p")
- $\Diamond p$ (possibly p) is equivalent to $\sim\Box\sim p$ (not necessary "not p")

K results from adding the following rule and axiom to the principles of propositional calculus:

- *Necessitation Rule*: If p is a theorem of K, then so is $\Box p$.
- *Distribution Axiom*: $\Box(p \longrightarrow q) \longrightarrow (\Box p \longrightarrow \Box q)$.

The system K is too weak to provide an adequate account of necessity. Many important axioms are not provable in K and they give rise to other well-known modal systems. For example, a desirable axiom is that if "p is necessary then p is true". The following are well-known elementary axioms that govern the necessity, iteration, or repetition of modal operators.

- $\Box p \longrightarrow p$
- $\Box p \longrightarrow \Box\Box p$
- $p \Rightarrow \Box\Diamond p$
- $\Box p \Rightarrow \Diamond p$
- $\Diamond p \Rightarrow \Box\Diamond p$

3.3.1 Deontic Logic

Deontic logic is the field of logic that is concerned with obligation, permission, and related concepts. Obligation and norms generally, seems to have a modal structure. The difference between "You must do this" and "You may do this" looks a lot like the difference between "It is necessary that" and "It is possible that". Deontic logic introduces the modal operators O for "it is obligatory that", P for "it is permitted that" and F for "it is forbidden that". These are the notions that have received more attention in deontic logic than others, like "it is non-necessary that" or "it is optional that". Permission and prohibition are defined by taking obligation as primitive operator:

- $P_A \leftrightarrow \sim O_{\sim A}$
- $F_A \leftrightarrow O_{\sim A}$

These assert that something is permissible if and only if its negation is not obligatory and forbidden if and only if its negation is obligatory.

Standard Deontic Logic (SDL) is the most cited and studied system of deontic logic, and one of the first deontic logics axiomatically specified. It builds upon propositional logic and it follows a simple and elegant Kripke-style semantics. SDL can be axiomatized by assuming that we have a language of classical propositional logic with an infinite set of propositional variables, the operators $\sim$, $\Rightarrow$ and O and by adding the following axioms:

- $O_A \Rightarrow P_A$
- $O_{(A \Rightarrow B)} \Rightarrow (O_A \Rightarrow O_B)$

The first axiom guarantees the consistency of the system of obligations by insisting that when A is obligatory, then it is permissible that A. The second axiom says that when it is obligatory that A implies B then if A is obligatory then B is obligatory too.

Norms are "patterns", "rules", "laws" that individuals or social groups obey or "regularities" that their behavior displays. Deontic Logic enables one to represent the norms and perform normative reasoning of human behavior by formalizing such concepts as obligation, permission and prohibition, and employing them in representation and reasoning. When the norms of an organization are identified, it is possible to predict and hence to collaborate with others in performing coordinated actions. Norms aims to be captured and represented in the form of deontic logic, in order to serve as a basis for coordinating intelligent autonomous agents to perform many activities and regulating the interactions among them.

3.3.2 Temporal Logic

The term Temporal Logic has been broadly used to cover all approaches to the representation of temporal information within a logical framework. It is also used to refer to the modal-logic type of approach introduced around 1960 by Arthur Prior under the name of Tense Logic and subsequently developed further by logicians and computer scientists. Applications of Temporal Logic include its use as a formalism for clarifying philosophical issues about time, as a framework within which to define the semantics of temporal expressions in natural language, as a language for encoding temporal knowledge in artificial intelligence, and as a tool for handling the temporal aspects of the execution of computer programs.

Temporal Logic contains four modal operators. The two basic operators are G for the future, which is read as "it will always be the case that", and H for the past, which is read as "It has always been the case that". The other temporal operators are F which is read as "It will at some time be the case that" and P which is read as "It has at some time been the case that". These operators are defined from the basic operators by using the following equivalences:

- $P_A \leftrightarrow \sim H_{\sim A}$
- $F_A \leftrightarrow \sim G_{\sim A}$

3.3.3 Epistemic Logic

Epistemic logic is a subfield of modal logic that studies reasoning about knowledge and belief. It provides insight into the properties of individual knowers and a means to model complicated scenarios involving groups of knowers. While epistemic logic has a long philosophical tradition dating back to Ancient Greece, it has many applications in philosophy, theoretical computer science, artificial intelligence and economics. It was C.I. Lewis who created the first symbolic and systematic approach to the topic, in 1912.

Syntactically, the language of propositional epistemic logic is simply a matter of augmenting the language of propositional logic with the basic epistemic operator K, which can be read as "it is known that". If there is more than one agent, the subscripts that are attached to the operator indicates whose agent the knowledge is represented. So the operator K_cA is read as "Agent *c* knows *A*". Epistemic logic can be extended to support the notion of common knowledge for a group of agents. For example, there is the operator E_GA which is read as "every agent in group G knows that A", and the operator C_GA which is read as "it is common knowledge to every agent in G that A".

3.4 Modelling Mental Attitudes and Normative Notions within Defeasible Logic

Recent work shows that defeasible logic is a nonmonotonic approach that can be extended with modal and deontic operators [40], [39], [41], [74], [38]. This report presents a computationally oriented nonmonotonic logical framework that deals with modalities, motivated by potential applications for modelling multi-agent systems and policies.

The logical framework deals with the following modalities:

1. *knowledge*
2. *intention*
3. *agency*
4. *obligation*
5. *permission*

This approach has many similarities with the BOID architecture that was described in subsection 3.2.6. As our system, BOID architecture is a rule-based framework where the motivational attitudes are represented as rules. The conflicts may arise among informational and motivational attitudes and the way these conflicts are resolved determines the type of the agent. Both systems capture the informational attitude of belief (or knowledge) and the external motivational attitude of obligation while the policy-based intention modality of our framework captures both the intention and desire components of BOID.

On the other hand, there many aspects which differentiate our system from BOID architecture:

- our framework is enriched with the additional notions of agency and permission;
- rules support the introduction of modalities, not only through the labelling of rules, but also explicitly in rule antecedents. Thus rules can also contain modalised literals, enriching in this way the expressive power of the logic;

- it supports the feature of rule conversion, in which the modality of one rule can be converted into another;
- the resolution of conflicts captures only some motivational attitudes. Conflicts may not appear among certain modalities;
- superiority relation is used between two single rules in order to devise specific policies.

Our logical framework is based on the approach of [40]. Similarly to BOID, it combines the two different perspectives of cognitive agents, based on the BDI architecture and the modelling of agent societies by means of normative concepts. In addition to this approach, we consider a fifth kind of modality, the deontic notion of *permission*.

3.4.1 Knowledge

Knowledge is the agent 's theory about the world. This is an epistemic notion. It corresponds to the information that an agent has about the environment and its beliefs about the world. What an agent believes may not necessarily be true and in fact may change in the future, when new information is added to the system. So knowledge has a defeasible nature.

3.4.2 Intention

Intention in our framework describes the agent 's policy. It is the policy-based intention and is based on Bratman 's classification of intention [17], [38]. Bratman classifies intention as deliberative, non-deliberative and policy-based. The difference between the three is the following: When an agent *i* has an intention of the form $INT_i^{t_1}\varphi, t_2$ (read as *agent i intends at* t_1 *to* φ *at* t_2) as a process of present deliberation, then it is called deliberative intention. On the other hand, if the agent comes to have such an intention not on the basis of present deliberation, but at some earlier time t_0 and has retained it from t_0 to t_1 without reconsidering it, then this intention it is called non-deliberative. The third case arises when intentions are general and concern potentially recurring circumstances in an agent 's life. Such general intentions comprise policy-based intentions, and are defined as follows: when the agent *i* has a general intention/personal policy to φ in circumstances of type ψ and *i* notes at t_1 that *i* is (will be) in a ψ-type circumstance at t_2, and thereby *i* arrives at an intention to φ at t_2.

From the above definition it can be noted that a policy-based intention is not deliberative, since there is no present deliberation concerning the action to be performed, but it depends on the circumstances. Neither is it a non-deliberative since it is not simply a case of retaining an intention previously formed. It is general intention that can be either periodic or circumstance-triggered.

Suppose that there is a TV agent whose main objective is to make recommendations of TV programs given a user 's preferences. The weekly update of the TV

guide is considered as an agent 's periodic policy-based intention. On the other hand, the agent has the circumstance-triggered policy-based intention to recommend programs that the user likes. The agent does not intend to recommend tv programs that happen at the same time and he choose to recommend the highest program according to the ratings. In such scenarios the general intentions need to be reconsidered. The general intention has a nonmonotonic nature and the agent does not intend to act no matter what. He intends an outcome only if he is sure that all the evidence to the contrary has been defeated. This comes in contrast with normal modal logic, where the agent intends all the consequences.

3.4.3 Obligation

Our framework also incorporates the intuition of obligation, according to the agent 's normative system. It is crucial aspect in the modelliing of autonomous agents based on cognitive as well as social models to formalize internal motivational attitudes such as intention and external motivational attitudes such as obligation. Intentions are viewed as internal constraints of an agent and obligations as external constraints. As constraint, obligation is nonmonotonic, which is a well-known character of deontic reasoning. An example that shows this intuition, by presenting two conflicting rules in a normative system, is the following: "One rule says that committing a harm implies responsibility, while an other rule says that acting in self-defence implies no responsibility".

The defeasibility of normative reasoning is a very well established phenomenon with many facets and many nonmonotonic systems have been proposed to capture it. Moreover, Nute [66], [67] has proposed to extend Defeasible Logic with deontic operators to capture normative phenomena, while [3] shows how regulations can be represented conceptually in Defeasible Logic.

3.4.4 Agency

We also enrich our framework by the notion of agency [30], which is described in a multi-modal logical setting. This is an aspect that differentiates this system if compared, for example, to BOID architecture. The intuition of agency has been studied in many contributions. Despite some well known limitations, modal agency is very general since it captures actions that are simply taken to be relationships between agents and states of affairs. It is also a flexible approach, since it allows for the easy combination of actions and concepts like powers, obligations, beliefs etc.

We focus on the idea of personal and direct action to realize a state of affairs, formalized by the well-known modal operator *E*. The formula $E_x A$ means that the agent *x* brings it about *A*. For example, $E_x A$ can have the form $E_{customer} makeorder(p)$, where *A* is an action predicate denoting an order, executed by the customer 's agent. Different axiomatizations have been provided for it but almost all include $E_x A \Rightarrow A$. This schema expresses the successfulness of ac-

tions that is behind the common reading of "bring about" concept. Other common axiomatizations are $\neg E_x \top$ and $(E_x p \wedge E_x q) \Rightarrow E_x(p \wedge q)$.

We follow the approach that was developed in [40], where the focus is on the intentional character of actions in order to handle the interaction between actions, intentions and other mental states. So we use the operator *Z* to express intentional actions and making clear the difference with the *E* operator that captures both intentional and unintentional actions. *Z* has the same properties as *E* plus the schema $Zp \Rightarrow Ip$. The intentional character of *Z* implies that is also a nonmonotonic notion.

3.4.5 Permission

In our work we enrich the logical framework of [40] with a fifth kind of modality, permission, which is a basic deontic operator. This component represents what an agent is permitted to do, according to his normative system and it is used commonly in policies describing (conditional) entitlements. For this reason we extend the multi-modal logical framework with permissions, in order to represent and reason with business rules and policies properly, in Semantic Web applications.

Permission is a notion that has been studied less frequently than obligation in deontic logic, because of the complexity that appears in it [15]. In our framework, in order to introduce permission, we studied the provability of this operator and follow the approaches that exist. The simplest way is to see an action as it is permitted if its prohibition is not derived from the code. This is a kind of negative permission. Another approach that was followed in [41], says that a permission for an action is derived, if a derivable defeater for this action exists, which is superior and defeats all the prohibitions of the action.

In our work we concentrate more on the approach that follows a kind of positive permission. According to this, a permission for an action is derived, if it is explicitly stated with the use of the permission operator in a rule mode or in a modalised literal, as happens with the other modal operators. In the next chapter and by presenting the metaprogram we will make transparent the three different approaches of introducing permissions in the formalism. However, our system is mainly based on dealing with the permission operator directly and stating it explicitly as every other operator.

Two important axioms of deontic logic that our system supports are that a permission is incompatible with a prohibition and something is prohibited if its negation is obligatory and vice versa. Therefore the deontic notion of prohibition is also supported indirectly, as it can be introduced through the obligation of a literal 's negation. In this way, the formalism supports the basic normative concepts of obligation, permission and prohibition and captures the requirements of business rules community [81], [16]. Modelling policies and business rules requires a formal specification language with high expressive power that includes deontic modalities.

3.4.6 A Defeasible Logic of Agency, Intention, Obligation and Permission

Commonly, both motivational attitudes and normative aspects are logically captured through the use of modal logics. Modal logics are extensions of classical propositional logic with some intensional operators. So modal logics are by definition monotonic. However as we know, classical propositional logic is not well suited to deal with real life scenarios, because the descriptions of real-life cases are, very often, partial and somewhat unreliable. Hence any modal logic based on classical propositional logic is doomed to suffer from not properly dealing with inconsistent information, that may easily arise in multi-agent and web environments.

As we have argued so far, reasoning about mental attitudes, like intention and agency, has a defeasible nature, and defeasibility is a key aspect for normative reasoning. Recent works by Thomason [84] and on BOID architecture showed that any system that aims at the integration of informational and motivational attitudes, like a multi-agent system, should be developed within a nonmonotonic setting. The two phenomena of mental attitudes and deontic notions, although are different and sometimes incompatible intuitions, they are subject to defeasibility. A number of strategies are provided in BOID that handle the interaction and solve the conflicts among motivational attitudes.

Defeasible Logic is the suitable formalism that can deal with these components and can capture their nonmonotonic behavior. The reason being ease of implementation [59], flexibility [4] and it is efficient [56]. A rule-based computationally oriented nonmonotonic formalism was developed that extends defeasible logic and represents and reasons with these modal operators. This proposed logic introduces and manipulates modalities and is flexible enough to deal with different intuitions about the interactions of the internal and external motivational attitudes. Extending defeasible logic with deontic operators captures the expressiveness that is required in modelling and reasoning with policies.

In our formalism, a *defeasible theory D* is a structure $D = (F, R^K, R^I, R^Z, R^O, R^P, >)$ where *F* is a finite set of facts, R^K, R^I, R^Z, R^O and R^P are, respectively, finite set of rules (strict, defeasible rules and defeaters) for knowledge, intentions, agency, obligations and permissions, and $>$ is the set of the superiority relationships between the rules of the theory.

Given an agent, *F* consists of the information the agent has about the world, its immediate intentions, its actions and its absolute norms, that consist of obligations and permissions. It covers the direct knowledge of an agent. The indirect knowledge comes from the form of rules. The main function of rules is to allow for the derivation of new conclusions and those conclusions can be new pieces of knowledge, new intentions, obligation etc. Accordingly we have divided the rules in rules for each modality. In particular, R^K corresponds to the agent 's theory and beliefs about the world, R^I, R^Z encode its policy and actions respectively, R^O and R^P encode the obligations and permissions that are defined from his normative system. Finally, the relation $>$ captures the strategy of the agent (or its preferences).

In order to correctly capture the mental and deontic notions we extend the signature of the logic with the modal operators. Thus if *l* is a literal then *agency(l)* is a modalised literal. A modalised literal is represented prefixed with a modal operator (agency, intention, obligation and permission). For example, a rule such as $Summertime \Rightarrow_Z VisitSpain$ allows to infer the agent 's intention to *visit Spain*. A modalised literal can be defined in a defeasible theory as a fact or as a rule 's conclusion, where the mode of the rule determines the mode of the conclusion. In the previous example, the rule mode is agency. An unmodalised literal belongs to the knowledge of the environment. Rules for knowledge does not produce modalised literals, but express the agent 's factual knowledge about the world. On the other hand, modalised literals can only occur in the antecedents of rules and are not permitted in consequents. This restriction is motivated from the fact that rules are meant to introduce the modalities.

In subsection 3.1.3 we presented the proof theory of defeasible logic and how a conclusion is represented in the form of a tagged literal. In a similar way, we can represent a conclusion of a defeasible theory *D* in our formalism as a tagged literal and it can have one of the following four forms:

- $+\Delta_X q$, which is intended to mean that *q* is definitely provable in modality *X* in defeasible theory *D*.
- $-\Delta_X q$, which is intended to mean that we have proved that *q* is not definitely provable in modality *X* in defeasible theory *D*.
- $+\partial_X q$, which is intended to mean that *q* is defeasibly provable in modality *X* in defeasible theory *D*.
- $-\partial_X q$, which is intended to mean that we have proved that q is not defeasibly provable modality *X* in defeasible theory *D*.

Though we argued about the defeasible nature of motivational attitudes, which are primarily incarnated by $\pm\partial_X q$, nothing prevents from having also mental states and deontic notions that are indisputable. For example, we are used to say that the conditional obligations are defeasible but we cannot reject the possibility that strict obligations occur in agent 's theory. The intuition behind the interpretation of the rules for obligation is that strict rules express hard constraints that cannot be violated, while defeasible rules represent soft constraints that can be violated in exceptional situations. For the same reason, our formalism supports indisputable and defeasible intentions and agencies. Finally, the distinction in the type of rules helps us to capture the difference between the knowledge that an agent has about his environment, which never changes, and his beliefs that may change when stronger evidence in the contrary is obtained.

3.4.7 Interaction Among Modalities and Agent Types

Our motivation is to model autonomous agents based on cognitive and social models. The result of this combination of perspectives is an account of agent 's be-

havior as a balance between mental states and external normative factors. As we mentioned before, our system is based on defeasible logic, because it is not only the suitable formalism for dealing with the nonmonotonic character of motivational attitudes, but it can also handle the $>$ relation and resolves conflicts among mental states and normative factors. As in the BOID architecture, this formalism defines agent types by stating conflict resolution types in terms of overruling between rules.

Agent types correspond to the different ways through which conflicts are detected and solved between different types of rules. A rule is attacked potentially by another rule with complementary literal in its head and different mode. Table 3.1 shows the relationship between the different modes of rules and for each kind of rule indicates all potential attacks on it. In this table we analyze ten combinations

$\Rightarrow_K q$ / $\Rightarrow_O \sim q$	$+\partial_K q$ / $-\partial_O \sim q$
$\Rightarrow_K q$ / $\Rightarrow_I \sim q$	$+\partial_K q$ / $-\partial_I \sim q$
$\Rightarrow_K q$ / $\Rightarrow_Z \sim q$	$-\partial_K q$ / $-\partial_Z \sim q$
$\Rightarrow_K q$ / $\Rightarrow_P \sim q$	$+\partial_K q$ / $-\partial_P \sim q$
$\Rightarrow_I q$ / $\Rightarrow_Z \sim q$	$-\partial_I q$ / $-\partial_Z \sim q$
$\Rightarrow_I q$ / $\Rightarrow_P \sim q$	$+\partial_I q$ / $+\partial_P \sim q$
$\Rightarrow_P q$ / $\Rightarrow_Z \sim q$	$+\partial_P q$ / $+\partial_Z \sim q$
$\Rightarrow_P q$ / $\Rightarrow_O \sim q$	$-\partial_P q$ / $-\partial_O \sim q$
$\Rightarrow_O q$ / $\Rightarrow_I \sim q$	type of agent
$\Rightarrow_O q$ / $\Rightarrow_Z \sim q$	type of agent

Table 3.1: Basic Attacks

in two columns. This is the way that all basic attacks are resolved for all type of agents. The first column presents the potential attack between two modalized rules. Since we refer to attacks between rules, we use defeasible rules, where the conclusion of a rule can be defeated by contrary evidence. In the second column we present the result of the potential attack. In case the attack fails, the conclusion of the first rule is defeasibly provable. That means that the mode of the first rule is not attacked by the mode of the second rule. For example knowledge is not attacked by obligation. On the other hand, in case of an attack, the conclusion of the first rule is not provable. For example, agency is attacked by intention and vice versa. In case a mode of a rule wins, that happens independently of the strength of the rules involved.

The general assumption is that we deal with realistic agents. In other words, beliefs (knowledge) override and attack all the other modal operators. The only exception to this view is that rules for agency attack rules for knowledge, since the former ones represent the intentional direct actions, the performance of which derives factual results. Mutual attacks also exist between intention and agency, since the latter are intentional in character and between obligation and permission, according to the basic deontic axiom of incompatibility between a permission and a prohibition. There is not any attack between permission and mental states. If an

agent has an intention or performs an intentional action, then there is no conflict if it is also permitted not to perform this action according to his normative system. On the other hand, the conflicts that arise in the interaction among internal constraints (intention and agency) and external constraints (obligation) must be settled. The way these potential attacks are resolved, determines the different type of agents, as shown in Table 3.2. It is worth noting that in each agent type we consider the case

$\Rightarrow_O q / \Rightarrow_Z \sim q$		$\Rightarrow_O q / \Rightarrow_I \sim q$		Agent Type
$+\partial_O q$	$+\partial_Z \sim q$	$+\partial_O q$	$+\partial_I \sim q$	Strongly independent
$+\partial_O q$	$-\partial_Z \sim q$	$+\partial_O q$	$+\partial_I \sim q$	Selfish saint
$+\partial_O q$	$-\partial_Z \sim q$	$+\partial_O q$	$-\partial_I \sim q$	Hypersocial
$-\partial_O q$	$+\partial_Z \sim q$	$-\partial_O q$	$+\partial_I \sim q$	Sinner
$-\partial_O q$	$-\partial_Z \sim q$	$-\partial_O q$	$+\partial_I \sim q$	Social sinner
$-\partial_O q$	$-\partial_Z \sim q$	$-\partial_O q$	$-\partial_I \sim q$	Hyperpragmatic

Table 3.2: Agent Types

that agency and intention are potentially attacked by obligation. It is meaningless the case where intention and agency conflict, because we mentioned this basic attack before, since rules for Z govern intentional actions.

We distinguish six agent types. A *strongly independent* agent is free to adopt intentions and to perform intentional actions in conflict with obligations. In a *hypersocial* agent, obligations override all the conflicting rules for action and intention, while in a *hyperpragmatic* agent, no derivation is possible. A *selfish saint* is the agent whose intention is in conflict with an obligation, but no intentional action to realize such content is performed. On the other hand, the *sinner* agent performs the action and the obligation is defeated. The *social sinner* has the intention, but the conflicting obligation is not derived and no violating action is performed.

3.4.8 Rule Conversion

Our formalism also supports the interesting feature of *rule conversion*, which affect the condition of applicability of the rules. According to this, we can obtain modalised conclusions by a certain modality through application of rules which can have different modes. The feature of converting from one type of conclusion into different one can be found in many formalisms and allows the combination of nonmonotonic and classical consequences. The conversions depend on the modality in which the premises of the rule are provable. In many cases, the conclusion of the rule inherits the modality of the antecedent.

For example, let $\neg open\_umbrella \Rightarrow_K wet$, which encodes the knowledge of an agent that knows that if he does not open the umbrella, then he will become wet. Now, if we know that the agent has the intention not to open his umbrella, represented as $I(\neg open\_umbrella)$, then we can conclude by rule conversion and from the previous rule that the agent has the intention to become wet. In other

words, since he knows that the consequence of his choice not to open the umbrella is to become wet, he intends to this result. So we conclude $+\partial_I wet$. Similarly if he has the obligation not to open his umbrella, represented as $O(\neg open\_umbrella)$ and that means that $+\partial_{O\neg open\_umbrella}$ has been proved, then it is obligatory for the agent to be wet ($+\partial_O wet$).

In Table 3.3 we present the rule conversions that are supported by our system. The first and second columns indicate the permitted modalities of the antecedent

X	Y	$\Rightarrow$	W
O	O	K	O
I	I	K	I
Z	Z	K	Z
Z	I	K	I
I	I	Z	I
O	O	Z	O

Table 3.3: Rule Conversions

of the rule, in order to satisfy the conditions for rule conversion. The third column specifies the mode of the rule and the fourth column the modality in which can be transformed the conclusion, if the previous conditions are satisfied. For example, according to the first case, a rule for knowledge can be used to directly derive an obligation, if the antecedent of the rule are provable in obligation. Or in the fourth case, a rule for knowledge can be used to obtain an intention, if its antecedent are provable in agency and intention.

Chapter 4

Translation Into Logic Programs

There are two different approaches for translating defeasible theories into logic programs, sharing the same basic structure:

- The translation of [6], [59], where a metaprogram is used to express defeasible logic in logic programming terms.
- The translation of [8], which makes use of control literals to translate a defeasible theory into logic program clauses.

It is an open question which is better in terms of computational efficiency, although we conjecture that, for large theories, the metaprogram approach is better since the latter approach generates a large number of program clauses. Therefore, we have adopted the metaprogram approach to formulate the proof theory of defeasible logic.

According to this, we translate a defeasible theory D into a logic program P(D), and we use a logic metaprogram that simulates the proof theory of the formalism that extents defeasible logic, to reason over the defeasible theory. The metaprogram was implemented in the logic programming language of Prolog.

4.1 Translation into Logical Facts

A defeasible theory is a structure $D = (F, R^K, R^I, R^Z, R^O, R^P, >)$ where F is a finite set of facts, R^K, R^I, R^Z, R^O and R^P are, respectively, finite set of rules (strict, defeasible rules and defeaters) for knowledge, intentions, agency, obligations and permissions, and $>$ is the set of the superiority relations between the rules of the theory. The basic predicates, which are used to represent a defeasibly theory, are translated as follows:

$fact(p)$. for each $p \in F$
$strict(r, m, p, [a_1, ..., a_n])$. for each rule $r : a_1, a_2, .., a_n \longrightarrow_m p \in R$
$defeasible(r, m, p, [a_1, ..., a_n])$. for each rule $r : a_1, a_2, .., a_n \Longrightarrow_m p \in R$

$defeater(r, m, p, [a_1, ..., a_n])$. for each rule $r : a_1, a_2, .., a_n \rightsquigarrow_m p \in R$
$superior(r, s)$, for each pair of rules such that $r > s$

The relationship between the proof tags of a defeasible theory *D* on one hand and the predicates *strictly* and *defeasibly* of the metaprogram M on the other is as follows:

$$D \vdash +\Delta_X p \text{ iff } M \vdash \textit{strictly(p, X)}$$
$$D \vdash -\Delta_X p \text{ iff } M \vdash \textit{not(strictly(p, X))}$$
$$D \vdash +\partial_X p \text{ iff } M \vdash \textit{defeasibly(p, X)}$$
$$D \vdash -\partial_X p \text{ iff } M \vdash \textit{not(defeasibly(p, X))}$$

4.2 Defeasible Logic Metaprogram

The translation into logic programming follows the approach described in [40], where the metaprograms that represent the extension of defeasible logic are presented. This approach also supports the definition of different types of agents, by handling the conflicts between the different types of modal rules, and the feature of rule conversion. Additionally to these features, our metaprogram supports the introduction of the permission operator and the representation in logic programming terms of the basic properties between the deontic notions that we mentioned in subsection 3.3.1.

There are six metaprograms, one for each agent type. We will present the metaprogram for the strongly independent type, which is free of conflicts between agency, intention and obligation and contains only the basic attacks between modalities. We will also illustrate how the clauses of this metaprogram can be modified in order to support different agent types with different policies in resolving conflicts. This metaprogram consists of clauses, introduced and described in the following subsections:

4.2.1 Supportive Rules

The first two clauses define supportive rules that are used to derive the provability of a literal. Strict and defeasible rules are supportive:

```
a1 : supportive_rule(Name,Operator,Head,Body) :-
     strict(Name,Operator,Head,Body).
a2 : supportive_rule(Name,Operator,Head,Body) :-
     defeasible(Name,Operator,Head,Body).
```

These predicates have similar structure to the rule predicates in metaprograms that have been developed for the propositional defeasible logic, with an additional argument. This is a modal operator that determines the mode of the rule. We mark a label in front of each clause, e.g. *a1*, to refer them easily in the examples of section 4.4.

4.2.2 Definitely Provable Literal

The next clauses define the definite provability: a literal is strictly (or definitely) provable in knowledge, if it is a fact and strictly provable in other modalities, if the corresponding modal literal is a fact. A modalised literal is represented as prefixed with a modal operator (agency, intention, obligation or permission). An unmodalised literal belongs to the knowledge of the environment. A literal is also strictly provable in a modality, if it is supported by a strict rule, with the same mode and the premises of which are strictly provable.

```
b1  : strictly(P,knowledge)     :- fact(P).
b2  : strictly(P,obligation)    :- fact(obligation(P)).
b3  : strictly(P,intention)     :- fact(intention(P)).
b4  : strictly(P,agency)        :- fact(agency(P)).
b5  : strictly(P,permission)    :- fact(permission(P)).
b6  : strictly(P,Operator)      :- strict(R,Operator,P,B),
                                   strictly(B).

b7  : strictly([]).
b8  : strictly([A1|A2])         :- strictly(A1),strictly(A2).
b9  : strictly(obligation(P))   :- strictly(P,obligation).
b10 : strictly(intention(P))    :- strictly(P,intention).
b11 : strictly(agency(P))       :- strictly(P,agency).
b12 : strictly(permission(P))   :- strictly(P,permission).
b13 : strictly(P)               :- strictly(P,knowledge).
```

4.2.3 Defeasible Provable Literal

The next clauses define defeasible provability: a literal is defeasibly provable in a modality, either if it is strictly provable in the same modality, or if the literal, for this modality, i) is consistent, ii) is supported by a supportive rule, and iii) there is not an undefeated applicable conflicting rule.

```
c1  : defeasibly(P,Operator)    :- strictly(P,Operator).
c2  : defeasibly(P,Operator)    :- consistent(P,Operator),
    supported(R,Operator,P),negation(P,P1),
    not(undefeated_applicable(S,Operator,P1)).

c3  : defeasibly([]).
c4  : defeasibly([A1|A2])       :- defeasibly(A1),
                                   defeasibly(A2).
c5  : defeasibly(obligation(A)):- defeasibly(A,obligation).
c6  : defeasibly(agency(A))     :- defeasibly(A,agency).
c7  : defeasibly(intention(A)) :- defeasibly(A,intention).
c8  : defeasibly(permission(A)):- defeasibly(A,permission).
c9  : defeasibly(P)             :- defeasibly(P,knowledge).
```

4.2.4 Consistent Literal

A literal is consistent in a modality, if its complementary literal is not strictly provable in the same modality and in any of the attacking modalities. In the previous

chapter and in Table 3.1 we show which are the basic attacks between modalities. Knowledge is a modality that is attacked only by agency, independently of the agent type. The next clause defines a literal 's consistency in knowledge for all agent types:

```
d1 : consistent(P,knowledge) :- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,agency)).
```

As we mentioned, agency and intention are mutually attacked, while knowledge attacks intention and obligation. Thus, in the metaprogram for strongly independent agent type, we should add the following clauses, which define a literal 's consistency in the other modalities:

```
d2 : consistent(P,obligation):- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,obligation)),
  not(strictly(P1,permission)).

d3 : consistent(P,permission):- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,obligation)).

d4 : consistent(P,intention) :- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,intention)),
  not(strictly(P1,agency)).

d5 : consistent(P,agency)    :- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,intention)),
  not(strictly(P1, agency)).
```

In case of an other agent type there are more attacks. In a hypersocial agent, obligation attacks agency and intention. Thus, we should add the predicate *not(strictly(P1,obligation))* to the clause that proves a literal 's consistency in intention and agency:

```
d6 : consistent(P,intention) :- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,intention)),
  not(strictly(P1,agency)),not(strictly(P1,obligation)).

d7 : consistent(P,agency)    :- negation(P,P1),
  not(strictly(P1,knowledge)),not(strictly(P1,intention)),
  not(strictly(P1,agency)),not(strictly(P1,obligation)).
```

4.2.5 Supported Literal and Rule Conversion

A literal is supported in a modality, if it is supported by a supportive rule with the same mode, the premises of which are defeasibly provable.

```
e1 : supported(R,Operator,P) :-
  supportive_rule(R,Operator,P,A),defeasibly(A).
```

The metaprogram also supports the feature of *rule conversion*, where the mode of a rule is converted into a different one. For example, as we can see in , we use the rule conversion to define that a rule for knowledge can be used to support a literal in obligation:

```
e2 : supported(R,obligation,P)            :-
     supportive_rule(R,knowledge,P,B),
     obligation_environment(B).
```

We use different predicates than the *defeasibly* predicate in the body of the clauses, in order to determine the defeasibly provability of the rule antecedent in the particular modalities required by the conversion. We call *environment* these literals in the antecedent of the conversion rule. In the above clause, we used the predicate *obligation_environment* to determine that the literals in the body of the rule must be provable in obligation. Thus we add the following clauses that define environment literals that are provable in obligation:

```
f1 : obligation_environment(A)            :-
     defeasibly(A,obligation).
f2 : obligation_environment(obligation(A)):-
     defeasibly(A,obligation).
f3 : obligation_environment([A1|A2])      :-
     obligation_environment(A1),obligation_environment(A2).
f4 : obligation_environment([]).
```

As we can see in Table 3.3, a rule for knowledge can also be used to transform a conclusion in an agency and intention, while a rule for agency can be used to obtain an intention and an obligation. We capture these cases by adding the following clauses in supporting a literal in a modality different than the mode of the supportive rule:

```
e3 : supported(R,agency,P)    :-
     supportive_rule(R,knowledge,P,A),agency_environment(A).
e4 : supported(R,intention,P) :-
     supportive_rule(R,knowledge,P,A),
     intention_agency_environment(A).
e5 : supported(R,intention,P) :-
     supportive_rule(R,agency,P,A),intention_environment(A).
e6 : supported(R,obligation,P):-
     supportive_rule(R,agency,P,A),obligation_environment(A).
```

The following clauses define environment literals that are provable in the modalities that are required in rule conversions:

```
f5 : agency_environment(A)               :-
     defeasibly(A,agency).
f6 : agency_environment(agency(A))       :-
     defeasibly(A,agency).
f7 : agency_environment([A1|A2])         :-
     agency_environment(A1),agency_environment(A2).
f8 : agency_environment([]).
f9 : intention_environment(A)            :-
     defeasibly(A,intention).
f10: intention_environment(intention(A)) :-
     defeasibly(A,intention).
f11: intention_environment([A1|A2])      :-
```

```
    intention_environment(A1),intention_environment(A2).
f12 : intention_environment([]).
f13 : intention_agency_environment(A)           :-
    defeasibly(A,intention).
f14 : intention_agency_environment((A))         :-
    defeasibly(A,agency).
f15 : intention_agency_environment(intention(A)):-
    defeasibly(A,intention).
f16 : intention_agency_environment(agency(A))   :-
    defeasibly(A,agency).
f17 : intention_agency_environment([A1|A2])     :-
    intention_agency_environment(A1),
    intention_agency_environment(A2).
f18 : intention_agency_environment([]).
```

The environment literal *intention_agency_environment* is used to define literals that are provable in intention or agency. It is required by the fourth row in Table 3.3, where the antecedent of a rule for knowledge is provable in agency and/or intention, in order to transform the conclusion in an intention mode.

4.2.6 Undefeated Applicable Rule

An undefeated applicable rule is a conflicting rule that is used to prevent the defeasible provability of a literal in a modality. A rule is undefeated applicable in a modality *M*, if it is an applicable rule in *M* or in a mode that attacks *M*, and it is not defeated by a conflicting rule in its mode *N* or in a modality that attacks *N*. For example, in a strongly independent agent type, a rule is undefeated applicable in agency, if it is a rule in intention (attacking modality) and it is not defeated by a conflicting rule in knowledge, agency (attacking modalities) and intention.

The following clauses define the undefeated applicable rules in different modalities:

```
g1 : undefeated_applicable(R,knowledge,P):-
    applicable(R,knowledge,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)).
g2 : undefeated_applicable(R,knowledge,P):-
    applicable(R,agency,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)),not(defeated(R,intention,P)).
g3 : undefeated_applicable(R,agency,P)   :-
    applicable(R,knowledge,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)).
g4 : undefeated_applicable(R,agency,P)   :-
    applicable(R,agency,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)),not(defeated(R,intention,P)).
g5 : undefeated_applicable(R,agency,P)     :-
    applicable(R,intention,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)),not(defeated(R,intention,P)).
g6 : undefeated_applicable(R,obligation,P) :-
    applicable(R,knowledge,P),not(defeated(R,knowledge,P)),
```

```
    not(defeated(R,agency,P)).
g7 : undefeated_applicable(R,obligation,P) :-
    applicable(R,obligation,P),
    not(defeated(R,knowledge,P)),
    not(defeated(R,obligation,P)),
    not(defeated(R,permission,P)).
g8 : undefeated_applicable(R,obligation,P) :-
    applicable(R,permission,P),not(defeated(R,knowledge,P)),
    not(defeated(R,obligation,P)).
g9 : undefeated_applicable(R,intention,P)  :-
    applicable(R,intention,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)),not(defeated(R,intention,P)).
g10 : undefeated_applicable(R,intention,P) :-
    applicable(R,knowledge,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)).
g11 : undefeated_applicable(R,intention,P) :-
    applicable(R,agency,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)),not(defeated(R,intention,P)).
g12 : undefeated_applicable(R,permission,P):-
    applicable(R,knowledge,P),not(defeated(R,knowledge,P)),
    not(defeated(R,agency,P)).
g13 : undefeated_applicable(R,permission,P):-
    applicable(R,obligation,P),not(defeated(R,knowledge,P)),
    not(defeated(R,obligation,P)),
    not(defeated(R,permission,P)).
```

In case of another agent type, like a sinner agent, where agency and intention attack obligation, we should add clauses that define applicable rules in intention and agency that are undefeated in obligation. The predicates *not(defeated(R,agency,P1))*, *not(defeated(R,intention,P1))* should also be added in each case we define that an applicable rule in obligation is undefeated in all the attacking modalities.

We can state an interesting aspect about the multi-modal logical formalism. Permission is the only exception in modalities in that there is no conflict between complementary literals. In particular, a literal is consistent in permission even though its complementary literal is definitely provable. An applicable rule in permission is undefeated even though it exists an applicable conflicting rule in permission, independently of the strength of the rules.

Thus in a defeasible theory, it is allowed to derive both p and $\sim p$ (definitely or defeasibly) in permission mode. This is an inconsistency that may arise in a defeasible theory, but it is an aspect that we must adopt in this formalism. It makes more sense if we allow an agent both to perform and not to perform an action according to his normative system. On the other hand it is inconsistent to both permit and prohibit an action. So our system supports the basic deontic axiom of incompatibility among permission and prohibition.

4.2.7 Applicable Rule

A rule is applicable in a modality, if it is any rule (supportive rule or defeater) for this modality that its premises are defeasibly provable. A supportive rule is also applicable even it has a different mode that can be converted to this modality with the feature of rule conversion, which we mentioned before. To capture this case, we define that a supportive rule is applicable in a modality, if a literal is supported in this modality by this rule:

```
h1 : applicable(R,Operator,P):-
     defeater(R,Operator,P,A),defeasibly(A).
h2 : applicable(R,Operator,P):-
     supported(R,Operator,P).
```

4.2.8 Defeated Rule

The next clause defines that an applicable rule is defeated in a modality, if exists an applicable superior conflicting rule in the same modality.

```
k1 : defeated(S,Operator,P):- negation(P,P1),
     applicable(R,Operator,P1),superior(R,S).
```

4.2.9 More Clauses

We introduce two more clauses in the metaprogram that capture the success of the agency operator:

```
m1 : strictly(P,knowledge)  :- strictly(P,agency).
m2 : defeasibly(P,knowledge):- defeasibly(P,agency).
```

We define the predicate negation to represent the negation of a predicate and evaluate the double negation of a predicate to the predicate itself.

```
m3 : negation(~(X),X):- !.
negation(X,~(X)).
```

4.2.10 Negative Permission Approach

As we mentioned in the previous chapter, apart from the basic approach of *positive* permission, we can formulate through a metaprogram the two other approaches in introducing the permission operator in our formalism. The simplest way to meet the requirements of permission consists in viewing the permission of an action as the negation of its prohibition. In this case, permission is implicitly expressed in the system. It corresponds to the non-derivability of an obligation. We use the predicate *permission* to describe this notion, as the modal operator *permission* is not used in a defeasible theory and stated explicitly any longer. Thus, if we have $-\partial_O p$, we can obtain *permission(p)*. The next clause define permissive derivability:

```
permission(P):-
     negation(P,P1),not(defeasibly(P1,obligation)).
```

As the permission operator is not directly introduced, we remove from the previous metaprogram all the cases that define the provability of permission. These are the clauses which have as head the predicates *strictly(P,permission)*, *strictly(permission(P))*, *defeasibly(permission(A))*, *consistent(P, permission)* and *undefeated_applicable(R,permission,P)*. We also remove all the potential attacks to obligation operator from permission, like stating *not(strictly(P1, permission))* and *not(defeated(R,permission,P))*.

This approach seems reasonable when there are no rules for obligation that support the complementary literals *p* and $\sim p$, or these rules exist but are inapplicable. On the other hand, if rules for obligation exist for these two literals, that are applicable but they defeat each other without the possibility of solving the conflict with a superiority relation, then it seems unreasonable to conclude both again *permission(p)* and *permission($\sim$p)*. For example, suppose that we have the two conflicting rules $r_1 : killer \Rightarrow_O punishable$ and $r_2 : mad \Rightarrow_O \neg punishable$ and that this conflict cannot be solved. In this case, it is unreasonable to derive both *permission(punishable)* and *permission($\neg$punishable)*.

4.2.11 Deriving Permissions Through Defeaters

The third approach follows the indirect introduction of permission in the formalism, as happens with the previous approach, but with a stronger statement that deals with the weakness of the previous approach. A literal *p* is permitted if a defeater in obligation for the same literal blocks all the attacks from conflicting prohibitions. But as we know, defeaters are used only to prevent a conclusion and not to derive it. Thus we define a special treatment of defeaters, following the approach from [41], where they are used as rules for supporting the derivation of a conclusion. A permission *p* is derived, if a defeater $\rightsquigarrow_O p$ exists, the premises of which are defeasibly provable and it is used not only to block the conflicting supportive rules for obligation, but it must also be superior from them in order to establish the conclusion *permission(p)*. To illustrate this, suppose there is a norm that forbids to U-turn at traffic lights unless there is a "U-turn permitted" sign. This scenario can be represented as follows:

$$r_1 :\Rightarrow_O \neg Uturn$$

$$r_2 : UturnSign \rightsquigarrow_O Uturn$$

The defeater in the second rule blocks the derivability of the first rule, but in order to conclude *permission(Uturn)*, we must add the superiority relation $r_2 > r_1$ in the defeasible theory.

The metaprogram has similar structure with that of the negative permission approach and it does not include clauses and predicates that define derivations and attacks from permission. It differs in the way that treats permissions and it is defined by the following clauses:

```
permission(P):- defeater(R,obligation, P, A),defeasibly(A),
    not(defeated_permission(P,permission,R)).
defeated_permission(P,permission,R):-
    negation(P,P1),supportive_rule(S,obligation,P1,A),
    defeasibly(A),not(superior(R,S)).
```

The defeater has similar treatment with the undefeated applicable rule from the metaprogram of positive permission. We use a new predicate *defeated_permission* to define a supportive rule in obligation (not a defeater) that is not inferior (even though it is not superior) and blocks the supportive defeater. As we can conclude, this approach is used preferably more in scenarios where the prohibitions are the normal case and permissions are the exceptions.

4.3 Arithmetic Capabilities in the Metaprogram

The rule language of defeasible logic does not support arithmetic and temporal operations, that are required in many applications, like in reasoning with business rules and regulations [3]. Since the modelling of policies and business rules was one of the basic motivations for our work, we augment the metaprogram with more facilities. This was achieved by embedding in the metaprogram the arithmetic capabilities that are offered by the Prolog engines.

Both the two logical systems (YAP and XSB), that we use as reasoning engines in our system, support these capabilities. They support numbers, both integers and floating-point numbers, that can be used as Prolog terms. Prolog offers binary operators in comparing Prolog terms and evaluating Prolog expressions. Therefore, we add the following clauses in the metaprogram, in order to support arithmetic comparison between terms:

```
strictly(greater(X,Y),knowledge)      :- X>Y.
strictly(equal(X,Y),knowledge)        :- X=Y.
strictly(different(X,Y),knowledge)    :- X\=Y.
strictly(greaterEqual(X,Y),knowledge):- X>=Y.
strictly(less(X,Y),knowledge)         :- X<Y.
```

For each type of comparison operation we define a corresponding predicate, to support this operation in the metaprogram. It is obvious that a conclusion of this kind of rule, the premises of which are a mathematical expression, never changes. So the conclusion of this rule is considered as definitely provable and belongs to the agent 's theory about the world. For example, if the variables *X* and *Y* are instantiated to numbers 8 and 5 respectively, then the Prolog Engine evaluates that the expression 8>5 is true and the metaprogram produces $+\Delta_{knowledge} greater(8,5)$ or that the literal *greater(8,5)* is definitely provable in knowledge.

Prolog supports evaluation of arithmetic expressions through the built-in predicate *is/2*. The predicate is(?X,+Y) succeeds if the result of evaluating the expression Y unifies with X. An arithmetic expression can use unary or binary operators. We add the following clauses in the metaprogram, in order to support several arithmetic operations between terms:

```
strictly(inc(X,Y),knowledge)        :-is(X,+(Y,1)).
strictly(add(X,Y,Z),knowledge)      :-is(Z,+(X,Y)).
strictly(equalMax(X,Y,Z),knowledge):-is(Z,max(X,Y)).
strictly(dec(X,Y),knowledge)        :-is(X,-(Y,1)).
strictly(sub(X,Y,Z),knowledge)      :-is(Z,-(X,Y)).
```

In the same way, for each type of arithmetic operation we define a corresponding predicate in the metaprogram. For example, if the variables *X* and *Y* are instantiated to numbers 8 and 5 respectively, then the Prolog Engine evaluates that the expression *is(Z,+(X,Y))* is true and the metaprogram produces $+\Delta_{knowledge} add(8, 5, 13)$ or that the literal *add(8,5,13)* is definitely provable in knowledge.

Prolog engines also offer modules that contain list manipulation predicates, as lists are useful utilities in calculations. A list is a Prolog structure that can be used to represent a sequence of Prolog terms. The two basic list predicates are *append/3* and *member/2*. The predicate *append(?List1,?List2,?List3)* succeeds if list *List3* is the concatenation of lists *List1* and *List2*. The predicate *member(?Element,?List)* succeeds if *Element* occurs in list *List*. The following clauses are added to the metaprogram in order to define list operations:

```
strictly(member(X,[X|List]),knowledge).
strictly(member(X,[Element|List]),knowledge):-
    strictly(member(X,List),knowledge).
strictly(append([],List,[List]),knowledge).
strictly(append([Element|List1],List2,
[Element|List1List2]),knowledge)            :-
    strictly(append(List1,List2,List1List2),knowledge).
strictly(not(X))                            :-
    not(strictly(X)).
```

4.4 Examples of Using the Metaprogram

We will describe some scenarios, in order to illustrate the way that the metaprogram is used in order to reason over the extension of defeasible logic with modal operators.

4.4.1 The Surgeon

This example is taken from [40]. Suppose that a drunk surgeon intends to operate a patient. He is aware that operating under the influence of alcohol will be an action with failed results. Besides, the law prescribes that people are responsible for causing damages as a result of carelessness. These two rules can be formalized using the formalism that extends defeasible logic as:

$$r1 : intention(operate), drunk \Rightarrow_Z fail$$
$$r2 : permanentDamages, agency(fail) \Rightarrow_O responsible$$

Since we want to reason and obtain conclusions from the defeasible theory, we need to translate the rules according to the metaprogram. The two rules are rewritten:

```
defeasible(r1,agency,fail,[intention(operate),drunk]).
defeasible(r2,obligation,responsible,
    [permanentDamages,agency(fail)]).
```

Suppose that we have a defeasible theory with the facts *F = (intention(operate), drunk, permanentDamages)* and *R = (r1,r2)*. By running the metaprogram and the logic programs which represent the facts and rules of the defeasible theory, we can conclude that *defeasibly(responsible,obligation)* or $+\partial_O responsible$. That means that the surgeon is responsible for causing permanent damage to a patient as a result of negligence. At first, from clause *c2*, we conclude $+\partial_Z fail$, as *agency(fail)* is supported by rule *r1* (clause *e1*), the premises of which are facts from the theory. Then the conclusion *obligation(responsible)* is supported by the rule *r2* (we use again clauses *c2* and *e1*), the premises of which are the previous conclusion *agency(fail)* and *permanentDamages*, which is a fact from the theory.

On the other hand, suppose that the law prescribes that if the patient will die without the operation and the surgeon is not on duty but he is the only person that is able to complete the required medical procedure, then the surgeon is not responsible for the results of this operation, independently of the surgeon 's situation during the operation. For this reason, we should add the following rule to the defeasible theory

$$r3 : patientDies, surgeonNotOnDuty \Rightarrow_O \neg responsible$$

and the superiority relation *r3* $>$ *r2*. We translate in logic programming as:

```
defeasible(r3,obligation,~(responsible),
    [patientDies,surgeonNotOnDuty]).
superior(r3,r2).
```

So if we obtain evidence to the contrary that the patient was dying and the surgeon was not on duty, then rule *r3* defeats *r2* (clause *k1*). We conclude the surgeon 's innocence $+\partial_O \neg(responsible)$, as *r3* is an undefeated applicable rule in obligation (clauses *h2* and *g7*).

4.4.2 The Prisoner 's Dilemma

We will illustrate the way that we use the formalism and especially the feature of different agent types, by formalizing the well-known prisoner 's dilemma from game theory, an example also taken from [40]. The classical prisoner 's dilemma is as follows:

Two people are arrested by the police for a major crime. The police does not have enough evidence to incriminate them, and, having separated both prisoners, visit each of them to offer the same deal: if one of them confesses the crime and

the other remains silent, the betrayer will be sentenced to one year and the silent partner receives the full 25-year sentence. If both stay silent, both prisoners have to serve for three years each. If each betrays the other, they have to serve for three years each. Each prisoner must make the choice of whether to betray the other or to stay silent. The best individual outcome for each prisoner is to confess the crime, while the best outcome according to the organization code of honor is not to confess and betray their partner.

The dilemma can be represented using the formalism as

$$r1 : committedCrime, arrestedWithPartner \Rightarrow_Z confess$$
$$r2 : committedCrime, arrestedWithPartner \Rightarrow_O \neg confess$$

or translated using the metaprogram clauses as:

```
defeasible(r1,agency,confess,
    [committedCrime,arrestedWithPartner]).
defeasible(r2,obligation,~(confess),
    [committedCrime,arrestedWithPartner]).
```

The dilemma is represented as a conflict between the intentional action of the prisoner to confess the crime and receive minor sentence and the obligation to the organization code of honor (his normative system) not to confess the crime. Actually the latter is a prohibition, as it is equivalent in the obligation not to act. The attack between agency and obligation is not basic and it is resolved by the agent type 's policy and in particular the clauses from 4.2.6, which determine when an applicable rule is undefeated in a modality for a particular agent type. For example, as it is defined from Table 3.2, a hypersocial prisoner, where obligation overrides agency, will stick with the code of honor and will not confess the crime. It will be concluded that $+\partial_O \neg confess$ and $-\partial_Z confess$ or that the predicate *defeasibly(∼(confess),obligation)* is true. Both these two literals are consistent (clauses *d2* and *d4*) and supported by the corresponding defeasible rules (clause *e1*), but rule *r2* is undefeated applicable and blocks the derivability of the agency to confess. On the other hand, a sinner prisoner, where agency overrides obligation, will confess the crime, giving in this way priority to his welfare. It will be concluded that $+\partial_Z confess$ and $-\partial_O \neg confess$. This is a violation of an obligation that does not imply its cancelation. The violated obligation is still in force but it does not make sense to deduce its consequent as a real obligation (or a prohibition). The prisoner is a case of legislator within the organization and he acts in a way that blocks the inference of $O\neg confess$. In case of a hyperpragmatic prisoner, obligation and agency are mutually attacked and only the superiority relations, that capture the strategy of the prisoner, among the rules can resolve this conflict and leads to a conclusion that detects his behavior in this dilemma. Thus, if the prisoner is hyperpragmatic and he has the preference *r2>r1* (clause *k1*), it means that in this particular situation, he chooses to obey to his normative code and not to confess the crime.

4.4.3 Umbrella Example

Let 's take the example that we presented in subsection 3.4.8, which uses the feature of rule conversion. The rule, that encodes knowledge of the agent about not opening the umbrella, is translated according to the metaprogram as:

```
defeasible(r1,knowledge,wet,~(open_umbrella)).
```

Adding to the agent 's defeasible theory the intention not to open the umbrella, is translated into the fact *intention(∼(open_umbrella))*. If we run the metaprogram and the logic programs that represent the facts and rules, we can conclude that $+\partial_I wet$ or *defeasibly(wet,intention)* is true. Rule *r1*, although is a rule for knowledge, it can be used to support the literal *∼(open_umbrella)* in intention, by using rule conversion (clauses *e4* and *f9*), as its premises are provable in intention. In the same way, if we add the obligation not to open the umbrella, represented as the fact *obligation(∼(open_umbrella))*, we conclude that $+\partial_O \neg wet$ or *defeasibly(˜(wet),obligation)* (from clauses *e2* and *f1*).

4.4.4 Weekend Example

The following example is borrowed from [23]. Here we show how we can represent and formalize a well-known example from BOID architecture, an approach that has several points of contact with our work. The scenario is the following:

My mother-in-law, who lives in Los Angeles, is in hospital and I am obliged to see her this weekend. On the other hand, i already have a plan to go to an important conference in New York, that is held also in this weekend. It is obvious that because of the distance, if i meet me brother-in-law i cannot attend the conference and vice versa.

The obligation of meeting the mother-in-law is formalized as:

```
defeasible(r1,obligation,meetMotherInLaw,weekend).
```

The intention to attend the conference as:

```
defeasible(r2,intention,attendConference,weekend).
```

We have the knowledge about the world that it is impossible to combine both meeting mother-in-law and attending the conference in a weekend and this is represented by the following rules:

```
defeasible(r3,knowledge,~(meetMotherInLaw),
    attendConference).
defeasible(r4,knowledge, ~(attendConference),
    meetMotherInLaw).
```

Thus, having the information that we are in the weekend (represented in the logic program as *fact(weekend)*), we use the formalism and the translations into the logic programming, in order to have an automate decision in choosing between

the obligation and the intention. At first, both the two literals *obligation(meetMotherInLaw)* and *intention(attendConference)* are supported by the first and second rule respectively (clause *c2*). But, the third and fourth rule are both applicable in intention and obligation respectively, by using rule conversion (clauses *e4* and *e2*). The conclusions depend on the type of the agent that defines the potential attacks. For example, in a hypersocial agent, the applicable rule *r4* in obligation attacks the intention rule *r2*. So i will obey to the obligations of my family and i will meet my mother-in-law in Los Angeles, concluding $+\partial_O meetMotherInLaw$, $+\partial_O \neg attendConference$ and $-\partial_I attendConference$. In case of a sinner agent, the applicable rule *r3* in intention attacks the obligation rule *r1*. So i will prefer my scheduled plan to attend the conference in New York, concluding $+\partial_I attendConference$, $+\partial_I \neg meetMotherInLaw$ and $-\partial_O meetMotherInLaw$.

4.4.5 Washington Conference

The following example is borrowed from [28], a research that is based again on BOID architecture. We will represent the following scenario:

I have an intention to attend a conference that will be held in Washington. The conference site is one of the most luxurious in Washington and all the rooms, close in this site, are quite expensive. Thus if i intend to go to Washington, I have the obligation to keep the company 's budget low and book a cheap room. On the other hand, if I have an intention to visit Washington, I have the intention to stay close to the conference site, because it facilitates my transfers.

We first represent the information that if someone stays in the conference site, he does not live in cheap room:

```
defeasible(r1,knowledge,~(bookCheapRoom),stayConferenceSite).
defeasible(r2,knowledge,~(stayConferenceSite),bookCheapRoom).
```

The obligation to book a cheap room is formalized as:

```
defeasible(r3,obligation,bookCheapRoom,
    intention(goWashingtonConference)).
```

My intention to book rooms near the conference site, if I have an intention to go to Washington is represented as:

```
defeasible(r4,intention,StayConferenceSite,
    intention(goWashingtonConference)).
```

Thus, having the information that I have the intention to attend a conference (represented as *fact(intention(goWashingtonConference))*, we will find if I obey to the obligation to book a cheap room or I prefer to intend stay close to the conference site and spend more money. Both are supported by the rules *r3* and *r4* respectively (clause *c2*). The first rules are applicable in intention and obligation (clauses *e4* and

e2). The type of the agent determines the derived conclusion, the obligation to book a cheap room $+\partial_O bookCheapRoom$ (*defeasibly(bookCheapRoom,obligation))* or the intention to stay in conference site $+\partial_I stayConferenceSite$ (*defeasibly(stayConferenceSite,intention))*.

4.4.6 Legal Reasoning

The following example is borrowed from [77]. In this scenario we will illustrate how our formalism represents and formalizes well-known examples from legal reasoning, a domain that is related with regulations and policies and incorporates deontic notions:

Article 2043 Italian Civil Code: A person is liable for damages he has intentionally caused, except for cases where the existence of justification causes is shown. These are the cases where the person accomplished the fact a) by self-defence, b) by a state of necessity, or c) he was incapable during the fact.

This article can be represented using the formalism as:

$r1 : agency(accomplished(X, Fact)), caused\_wrongful(Fact, Damage) \Rightarrow_O liable(X, Damage)$
$r2 : self\_defence(X, Fact) \Rightarrow_O \neg liable(X, Damage)$
$r3 : under\_necessity(X, Fact) \Rightarrow_O \neg liable(X, Damage)$
$r4 : incapable\_during(X, Fact) \Rightarrow_O \neg liable(X, Damage)$
$r2 > r1, r3 > r1, r4 > r1$

or translated using the metaprogram clauses as:

```
defeasible(r1,obligation,liable(X,Damage),
    [agency(accomplished(X,Fact)),
     caused_wrongful(Fact,Damage)]).
defeasible(r2,obligation,~(liable(X,Damage)),
    self_defence(X,Fact)).
defeasible(r3,obligation,~(liable(X,Damage)),
    under_necessity(X,Fact)).
defeasible(r4,obligation,~(liable(X,Damage)),
    incapable_during(X,Fact)).
superior(r2,r1).
superior(r3,r1).
superior(r4,r1).
```

This is a case where we model a norm and its exception-provisions as defeasible rules with higher priority. In this certain situation, a particular normative statement is blocked and does not apply. For this reason we use the mode *obligation* for the modality of these rules 's conclusions, as they define a person 's responsibility for harmful actions, according to the normative system of law. We also model the intentional action to accomplish this fact by using in the antecedent of the rules the literal *accomplished(X,Fact)*, modalised by operator *agency*.

Suppose now that we have the following case: *Mary, during a quarrel with her husband Mark, has defended herself by throwing against him a dish. This dish belongs to Mark 's valuable collection of pottery and the result of the throw was its break.* These facts are represented as:

```
fact(agency(accomplished(Mary,ThrowDish))).
fact(caused_wrongful(ThrowDish,BreakDish)).
fact(self_defence(Mary,ThrowDish)).
```

We will use the rules from article 2043, along with the facts from this case, in order to automatically decide if Mary is liable for breaking Mark 's dish. In this defeasible theory, conflicting rules *r1* and *r2* are both applicable (clause *c2*), but *r2* has higher priority (clause *k1*). So we conclude that $+\partial_O \neg liable(Mary, BreakDish)$ or differently, that Mary is not responsible for this fact.

The following rule encompasses the notion of permission:

Article 2033 Italian Civil Code (undue payment): A person who has accomplished a payment that was not due has the right to claim back what he has paid.

This article is represented as a rule for permission:

$accomplished\_payment(X, Y), undue(Y) \Rightarrow_P claim\_back(X, Y)$

Chapter 5

Implementation Architecture

5.1 Overview of the Architecture

Our nonmonotonic rule-based system supports reasoning in defeasible logic, extended with modalities. It integrates with the Semantic Web, as it reasons with the standards of RDF and RDF Schema. It provides automated decision support, when running a specific case with the given logic programs and ontological knowledge to get a correct answer. Figure 5.1 presents the overall architecture of our system, which consists of different modules.

The system works in the following way: A user imports its rules and facts as logic programs. They follow the structure of the extended metaprograms with modalities, which translate defeasible theories into logic programs and perform defeasible reasoning, as we described in chapter 4. The user has the ability to choose the agent type metaprogram, that determines the way that the attacks between modalities are resolved during defeasible reasoning. In this case, the corresponding logic metaprogram is loaded to the system 's reasoning engine. The user can select between two choices for reasoning engines: YAP [92] and XSB [91].

Facts may come not only directly from logic programs, but also from the Web and in particular from RDF documents. The system has the additional functionality that treats RDF data as facts of the user's defeasible theories, in order to be processed by the rules. The RDF/S documents are retrieved from the Web, and validated by the Semantic & Syntactic Validator, before being loaded to the system. Then the system communicates with an instance of SWI-Prolog [82] system. It employs the SWI-Prolog Semantic Web library to load the syntactically and semantically valid RDF/S documents and translate them into RDF triples. The triples that have come from RDF data are translated into Prolog facts, and then are passed to the Reasoning Engine. The RDF Translator also translates triples that have come from RDFS documents into logical rules that capture the RDF Schema semantics. These Prolog rules are passed to the Reasoning Engine and further Prolog facts are entailed.

The system provides a Graphical User Interface (GUI). By interacting with

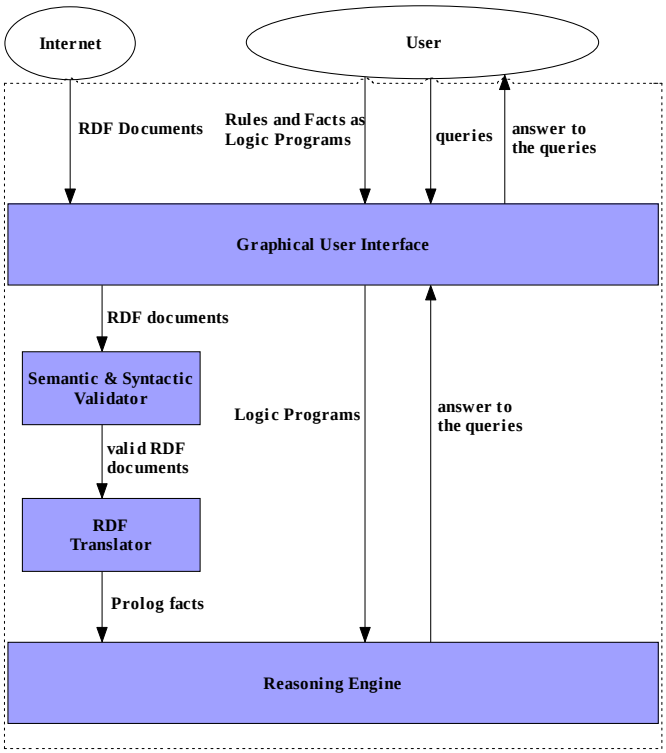

Figure 5.1: The Overall Architecture of our System.

the GUI, the user can import logic programs, load RDF/S ontologies and query the system. On the other hand, the system displays messages to the user through the GUI. The system also employs the Java programming library of InterProlog [49], an interface that provides access to the Prolog systems of YAP and XSB, in passing the logic programs and processing the user 's queries. InterProlog also provides access to the Prolog system of SWI-Prolog, which contains the Semantic Web library that translates RDF/S data, passed from the system, in Prolog facts and rules.

The reasoning engine compiles the metaprogram, which corresponds to the agent type we use, and the logic programs, which represent the rules and contain the ontological knowledge. Logic programs must have valid Prolog syntax, otherwise the system informs the user about the errors in syntax. The reasoning engine also evaluates the answers to user's queries. If these queries are syntactically cor-

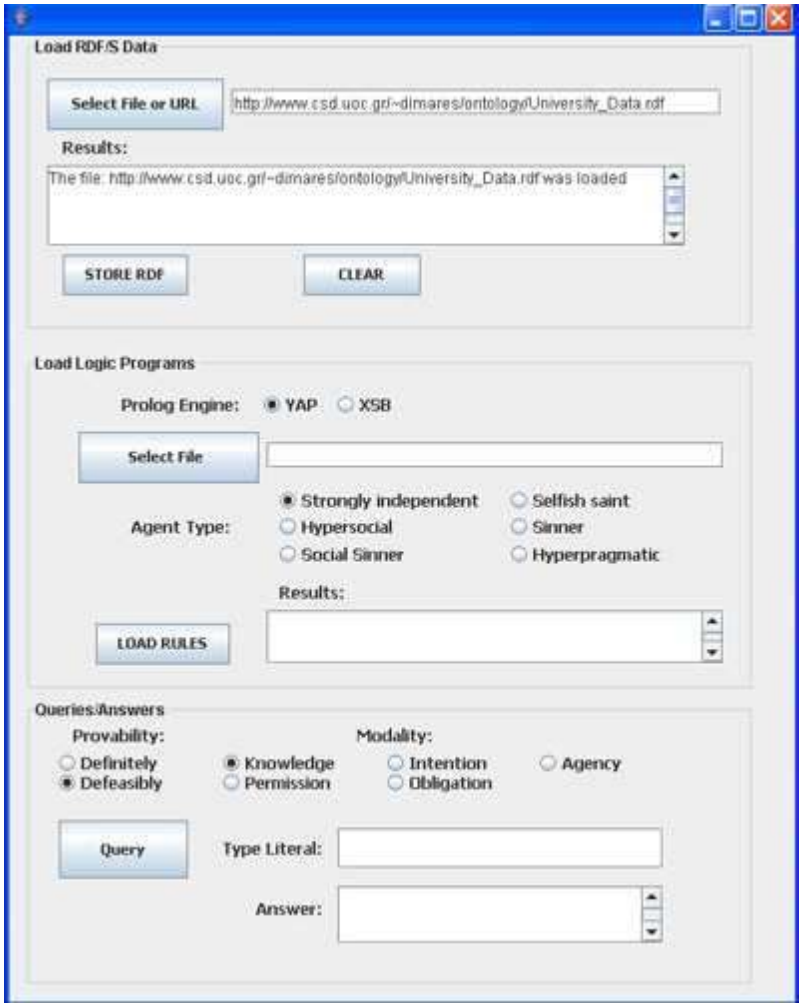

Figure 5.2: System 's Graphical User Interface

rect then are applied to the compiled programs. Otherwise the system informs the user about the errors in syntax.

5.2 Graphical User Interface

The system provides a Graphical User Interface based on Java Foundation Classes (Swing), that allows the user to interact with the underlying system. A screenshot of our system is presented in figure Figure 5.2. The interaction with the graphical user interface can be distinguished in three main tasks: a) loading RDF documents, b) loading logic programs and c) querying the system. Each task takes a different component in GUI 's panel.

5.2.1 Loading RDF Documents

User can select RDF/S documents, in order to be loaded to the system. He inserts the URL of the location where the document exists, or he selects to store local RDF documents, by inserting the directory where the file exists.

If the user pushes the "STORE RDF" button, he imports an RDF(S) document to the system. This file is firstly sent to the semantic & syntactic validator, where it is checked and if it is valid, then it is passed to the RDF Translator, where it is translated into Prolog facts or rules. This Prolog file is sent and loaded to the reasoning engine and then a message is displayed to the user, that informs him that the RDF file was valid and loaded to the system, as Figure 5.3 shows. The button

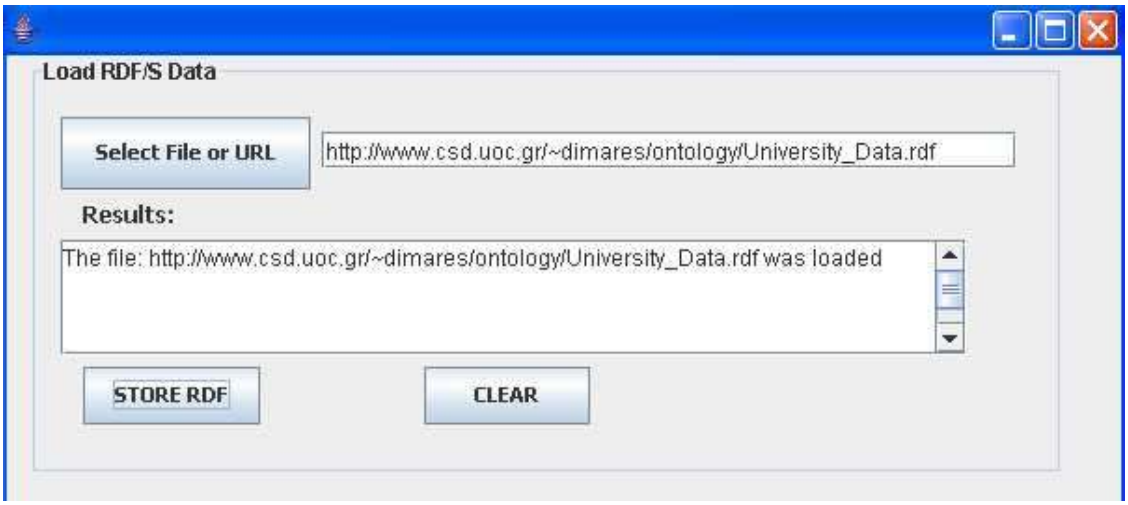

Figure 5.3: Load RDF/S Data

"CLEAR" is used when the user prefers to clean the messages that are displayed in this area. In case where he inserts an invalid RDF document, then the validator notifies the system and the appropriate message is displayed to the user, as is shown in Figure 5.4. This message informs the user about the errors of the invalid file, that was not loaded to the system.

5.2.2 Loading Logic Programs

By using the middle component, user can import Prolog files into the system. He can choose the Prolog system, that will be used as reasoning engine (YAP or XSB), and the agent type metaprogram, according to the way he prefers the resolution among modalities. In importing logic programs, the user types the directory where exists the local Prolog file, or he presses the button "CLEAR" to find it. By pressing the button "LOAD RULES", the file is sent to the reasoning engine. If it has valid syntax, then the logic program is loaded and the system informs the user through a message. In Figure 5.5, user has selected YAP as the Prolog system, strongly

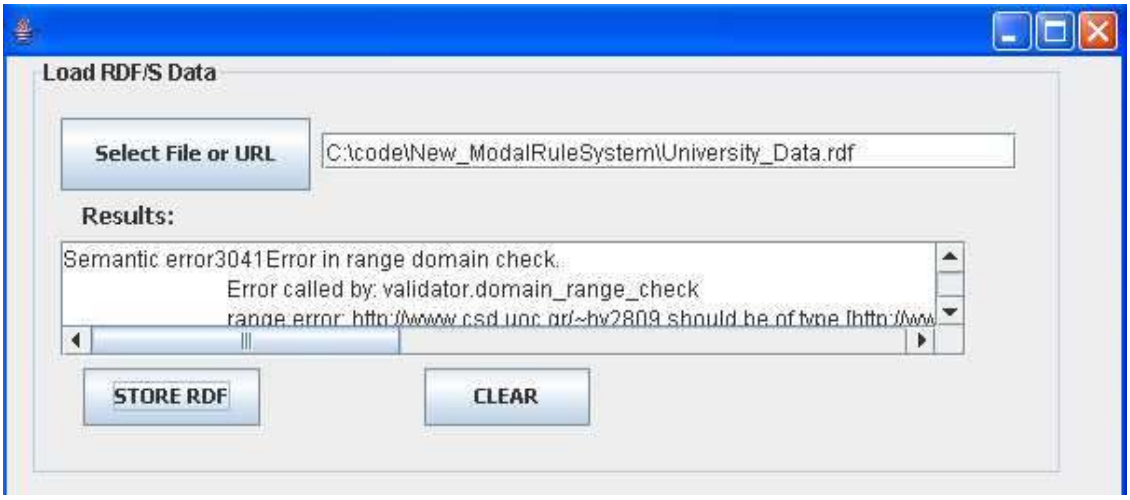

Figure 5.4: Invalid RDF File

independent as the agent type, and he loads successfully a Prolog file. On the

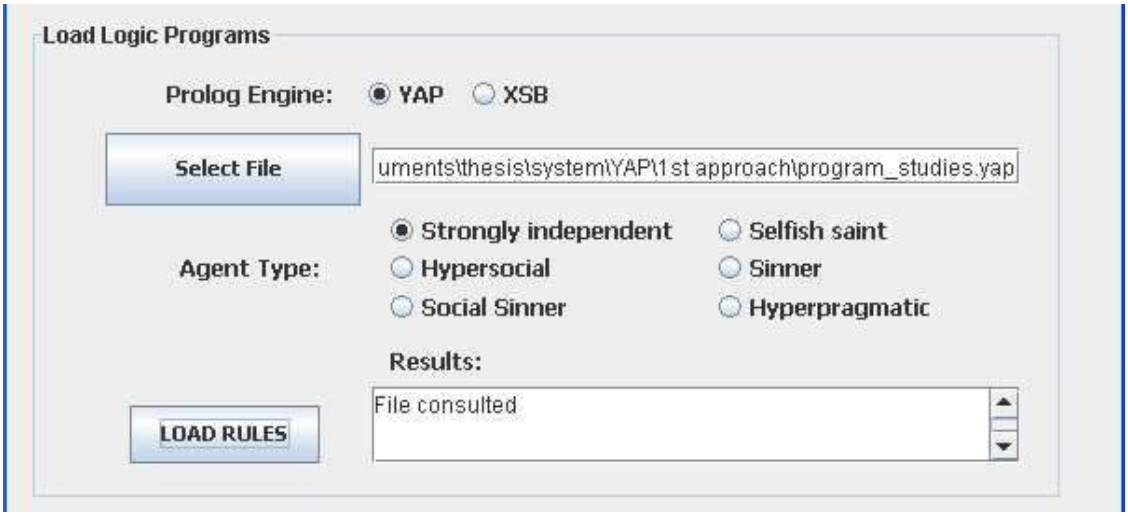

Figure 5.5: Logic Program Imported

other hand, the user 's logic program may be not loaded to the system, for reasons like due to error in Prolog Syntax, incompatible type of file (e.g. if XSB is the reasoning engine, then only ".P" files are loaded), wrong path directory etc. Then the system displays the error message to the user. Figure 5.6 shows a case where the user imports an XSB Prolog file, which is wrong in syntax and it is not loaded.

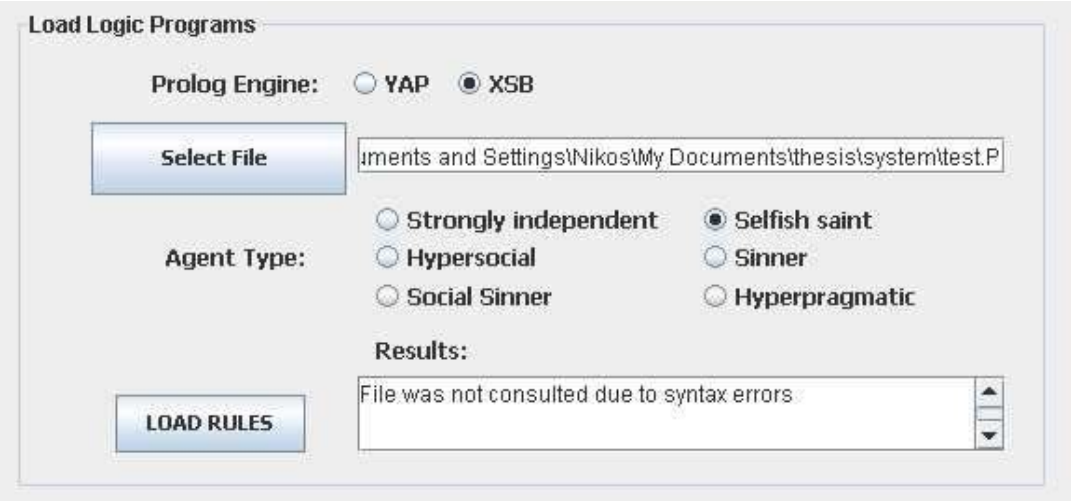

Figure 5.6: Invalid Prolog File not loaded

5.2.3 Querying the System

The last component of the GUI offers a way of interaction between the user, who inserts the queries, and the system, which returns the answers. Our system facilitates the typing of queries, by offering choices in the query 's provability, as definitely or defeasibly, and choices in the modality of the query 's literal, as knowledge, intention ,obligation etc. The user must only insert the literal of the query.

If the query has valid Prolog syntax, it is applied to the compiled Prolog files, which consist of the logical metaprogram and logic programs that the user may have inserted before, directly or from translated RDF documents. Then the reasoning engine evaluates and returns the answer to the system, which is a positive "yes" or a negative "no".

Figure 5.7 shows such a case. The user inserts the literal "enroll('Nikos Dimaresis','cs-280',fall,2007)" and the choice for query 's provability is *defeasibly*, while the modality is *permission*. Thus, the query is translated into the Prolog query "defeasibly(enroll('Nikos Dimaresis','cs-280',fall,2007), permission)" and it is sent to the reasoning engine. If the user inserts a literal with false syntax, then the reasoning engine returns to the system the error message about the invalid query and this is displayed to the user. Figure 5.8 shows a case, where the user inserts the literal "enroll('Nikos Dimaresis','cs-280',fall,2007", in which he has forgotten to close the parenthesis. This is not a valid Prolog query syntax and the system displays the corresponding message.

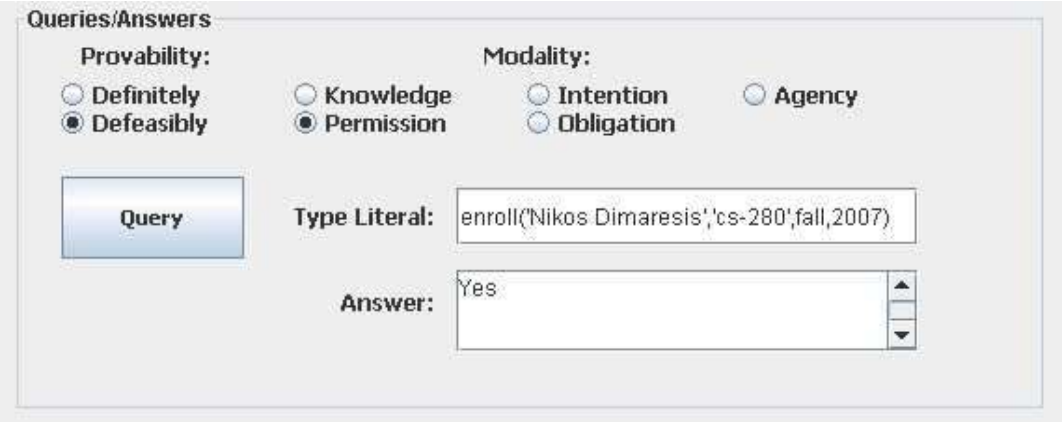

Figure 5.7: User Queries the System

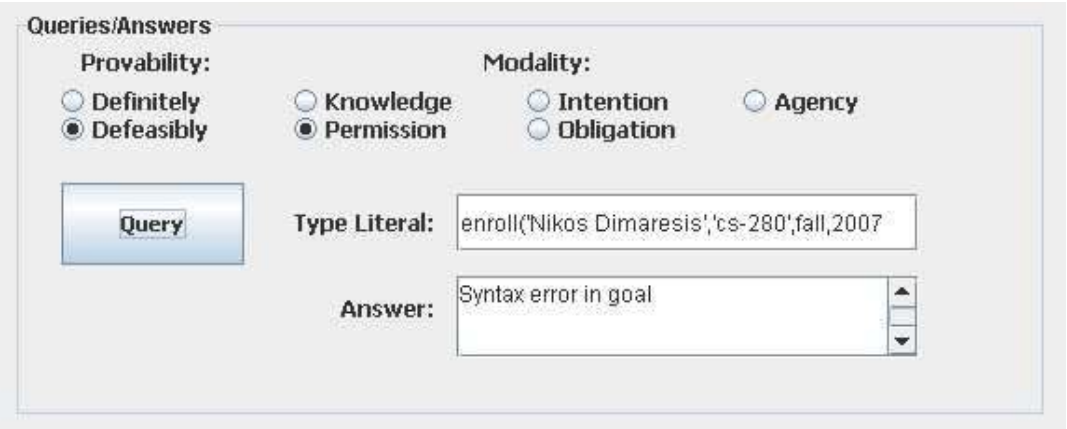

Figure 5.8: Error in Query 's Syntax

5.3 The Semantic & Syntactic Validator

This module of semantic & syntactic validator is a Java library that embeds the 3.0 version of VRP [87] and checks the RDF documents before being loaded into the system. The ICS-FORTH Validating RDF Parser (VRP v3.0) is a tool for analyzing, validating and processing RDF documents. This parser is part of the ICS-FORTH RDFSuite [83], a platform that comprises software tools that addresses the need for effective and efficient management of large volumes of RDF descriptions and schemas, as required by real-scale Semantic Web applications.

VRP analyses syntactically the statements of a given RDF/ XML file according to the RDF Model & Syntax Specification. It also checks whether the statements contained in both RDF schemas and resource descriptions satisfy the semantic constraints derived by the RDF Schema Specification (RDFS). The semantic constraints that are applied in RDF descriptions are:

- Class Hierarchy Loops
- Property Hierarchy Loops
- Domain/Range of SubProperties
- Source/Target Resources of properties
- Types of Resources

VRP reports to our system whether the RDF file is valid and sends diagnostic messages in case of several kinds of errors, like syntax, semantic and system errors. The error reporting is one the parser 's main features and the system uses the GUI to display these messages to the user. Otherwise informs him about the successful storage of the valid RDF document. VRP is based on standard compiler generator tools for Java, namely CUP (0.10j) and JFlex (1.3.5) similar to YACC/LEX, which ensure good performance when processing large volumes of RDF descriptions.

5.4 The RDF Translator

The RDF translator is the module of our system that translates valid RDF and RDF Schema statements into Prolog facts and rules. This transformation allows the RDF/S information to be processed by the rules provided by the user. Therefore the system supports reasoning with RDF/S ontologies and integrates the Semantic Web standards of RDF/S with our rule language that extends Defeasible Logic with modal and deontic operators.

In processing RDF/S documents we employ the SWI-Prolog Semantic Web library, that we mentioned in subsection 2.5.4. By using this library, when we load an RDF/S document in SWI, at first it is transformed into an intermediate format, where RDF triples are represented using predicates in the form of *rdf(Subject, Predicate, Object)*.

In the next step SWI uses several built-in predicates to process these triples and tranform them in a more suitable format. In case where RDF data has been loaded, RDF triples are transformed further into the format *Predicate(Subject, Object)*. The namespaces of all the elements are cut, while triples with *rdfs:comment* elements are removed. Then the triple is transformed in the format of fact in defeasible logic, that is *fact(Predicate(Subject, Object))*. When the processing of RDF triples is finished, all the facts are written in a new file, which is sent back to the system. Then the system sends this logic program to the reasoning engine in order to be loaded.

For example, suppose that we load the RDF example, from section 2.2, in SWI-Prolog. The following triple is produced:

```
rdf('http://www.csd.uoc.gr/~hy467',
    'http://www.csd.uoc.gr/~dimares/ontology/
    University.rdfs#isTaughtBy','Grigoris Antoniou').
```

This triple is transformed into the following fact:

```
fact(isTaughtBy('http://www.csd.uoc.gr/~hy467',
    'Grigoris Antoniou')).
```

In addition, SWI processes RDF Schema information, by translating RDF triples into Prolog facts and rules, by following the rules that capture the semantics of RDF Schema constructs:

```
a : C(X)    :- rdf : type(X,C).
b : C(X)    :- rdfs:subClassOf(Sc,C),Sc(X).
c : P(X,Y)  :- rdfs : subPropertyOf(Sp,P),Sp(X,Y).
d : D(X)    :- rdfs:domain(P,D),P(X,Z).
e : R(Z)    :- rdfs:range(P,R),P(X,Z).
```

The first rule defines the core property *rdf:type*, which defines the relationship between a resource X and a class C. It declares that the resource is an instance of this class. When loading an RDF document in SWI, the RDF triples of the form *rdf(Subject, rdf:type, Object)* are transformed in facts of the form *fact(Object(Subject))* and then are passed to the reasoning engine. Suppose the following triple is produced:

```
rdf('http://www.csd.uoc.gr/~hy467',rdf:type,
    'http://www.csd.uoc.gr/~dimares/ontology/
     University.rdfs#Course').
```

This triple is transformed into the following fact:

```
fact(course('http://www.csd.uoc.gr/~hy467')).
```

The second rule defines the core property of *rdfs:subClassOf*, which defines hierarchy in classes. It defines that if class Sc is subclass of class C, then each instance of Sc is also an instance of C. When loading an RDFS document in SWI, the RDF triples of the form *rdf(Subject, rdfs:subClassOf, Object)* are transformed in rules of the form *fact(Object(X)):- fact((Subject(X))* and then are passed to the reasoning engine.

For example, suppose we load the RDF Schema document, the statements of which, along with RDF data, are represented in the graph of Figure 2.4. Then SWI produces the following triple:

```
rdf('http://www.csd.uoc.gr/~dimares/ontology/
    University.rdfs#Professor',rdfs:subClassOf,
    'http://www.csd.uoc.gr/~dimares/ontology/
    University.rdfs#Lecturer').
```

This triple is transformed into the following rule, that captures the semantics of subclass relationship:

```
fact(lecturer(X)):- fact(professor(X)).
```

This Prolog rule is returned to the system and it is sent and loaded to the reasoning engine. It allows to infer additional facts from the facts that were initially loaded. Thus, after loading RDF data in SWI, from the initial RDF triple

```
rdf('Grigoris Antoniou',rdf:type,'http://www.csd.uoc.gr/
    ~dimares/ontology/University.rdfs#Professor').
```

and after translating the statements into Prolog facts and rules, the following fact is also entailed:

```
fact(lecturer('Grigoris Antoniou')).
```

The third rule defines the core property of *rdfs:subPropertyOf*, which defines hierarchy in properties. It defines that if property Sp is subproperty of property P and X,Y are respectively the subject, object of Sp, then X is also a subject of P, and Y is an object of P. When loading an RDFS document in SWI, the RDF triples of the form *rdf(Subject, rdfs:subPropertyOf, Object)* are transformed in rules of the form *fact(Object(X,Y)):- fact((Subject(X,Y))* and then are passed to the reasoning engine.
For example, suppose SWI produces the following triple:

```
rdf('http://www.csd.uoc.gr/~dimares/ontology/
    University.rdfs#isTaughtBy',rdfs:subPropertyOf,
    'http://www.csd.uoc.gr/~dimares/ontology/
    University.rdfs#involves').
```

This triple is transformed into the following rule, that captures the semantics of subproperty relationship:

```
fact(involves(X,Y)):- fact(isTaughtBy(X,Y)).
```

This Prolog rule is returned to the system and it is sent and loaded to the reasoning engine. It allows as to infer additional facts from the facts that were initially loaded. Thus, after loading the RDF example from 1.2 in SWI, and after translating the statements into Prolog facts and rules, the following fact is also entailed:

```
fact(involves('http://www.csd.uoc.gr/~hy467',
    'Grigoris Antoniou')).
```

The fourth and fifth rule capture the relationship between a property and its domain and range. The subject of a property P must belong to the class which is specified by the domain D of the property, and the object of a property P must belong to the class which is specified by the range R of the property. Actually SWI does not process the RDF triples that contain the properties *rdfs:domain(P,D)* and *rdfs:range(P,R)* and it does not transform them in Prolog rules. VRP parser checks the semantic constraint of source and target of properties in an RDF document, before this file is sent to the RDF translator. In case where an RDF file contains a statement where the subject or object is not instance of the domain or range class of the property respectively, then VRP reports the system about this error and the file is not sent to the RDF translator.

5.5 InterProlog

InterProlog is an open-source programming library for developing Java + Prolog applications. It is proposed as a bridge between Java and Prolog, promoting an integration between logic and object-oriented layers. InterProlog implements a bidirectional predicate/method calling between both languages, by mapping Java objects into Prolog terms and vice-versa. The communication between a Java application and a Prolog system is done through TCP/IP sockets or the Java Native Interface. Prolog processes are launched in the background, outside the Java Virtual Machine. InterProlog supports the Prolog systems of XSB, YAP and SWI through the same API.

5.6 YAProlog

YAP (*Yet Another Prolog*) is one of the two choices that can be used as reasoning engine and it is widely considered one of the fastest available Prolog systems. It is a high-performance Prolog compiler with several optimizations and the whole system is written in C. YAP obtains performance comparable to or better than commercial Prolog systems. It also provides many built-ins, including I/O functionality, data-base operations, modules and arithmetic operations, as the latter are embedded in the metaprogram (as we showed in chapter 4) and are required in many applications. Actually YAP is more suitable than XSB in applications with large defeasible theories, with many facts and rules, which require Prolog systems that offer small execution times.

5.7 XSB

XSB is an open source commercial logic programming system that extends Prolog with new semantic and operational features, mostly based on tabled resolution and HiLog. Tabled resolution is useful for recursive query computation, allowing programs to terminate correctly in many cases where Prolog does not. Users interested in several applications like Parsing, Program Analysis, Temporal Reasoning etc may benefit from XSB. HiLog is a standard extension of Prolog, permitting limited higher-order logic programming in which predicate symbols can be variable or structured and that allows their unification.

The main reason for selecting XSB as the system 's reasoning engine is the need for a Prolog system that supports the well-founded semantics. The choice of well-founded semantics has the advantage that it offers low computational complexity and it can detect cycles in the theories, without running into infinite loops. Under this approach, the translation of a defeasible theory D into a logic program *P(D)* has a certain goal: to show that

$$p \text{ is defeasibly provable in } D \Leftrightarrow$$
$$p \text{ is included in the Well-Founded model of } P(D)$$

XSB supports well-founded semantics of logic programs through the use of *sk_not* operator instead of *not* and the use of tabled predicates. This negation operator allows for the correct execution of logic programs with well-founded semantics and deals with cyclic theories. The logical metaprograms that were described in chapter 4, remain the same, with the only difference that the *not* operator is replaced by *sk_not*. In order to declare that the predicates are tabled, we include in each metaprogram the directive *:- auto_table*.

Chapter 6

A System Use Case: University Regulations

In this chapter we describe a use case that shows how our system works and interacts with the user 's queries. This is an example from a specific application, the modelling of a variety of university regulations from the Department of Computer Science at the University of Crete. Our system uses the formalism that extends defeasible logic with modalities and especially with deontic operators, in order to apply logic modelling to the representation and use of regulations. Thus we describe a use case that shows the functionality of our system in providing automated support for reasoning with regulations and integrating with the Semantic Web.

6.1 Modelling Regulations

Regulations are the type of business rules that codify how products must be made and process should be performed. They are a wide-spread and important part in the organization and functioning of society in general, and business in particular. In an environment of increasing complexity of, and change in, regulation, mainly due to technological change and the current trend towards globalisation, automated support for reasoning with regulations is becoming necessary.

Regulations are one of the many domains where business rules are applied. In this case, business rules are used by an organization either to ensure compliance with regulations, or to make decisions under regulations. Therefore, reasoning with regulations is compatible with one of the main motivations of our work, which is modelling of policies and business rules.

The use case is based on previous logical approaches in legal reasoning [70], [34], [86], [61] and in formalizing regulations [3] and business rules [1]. Formal systems offer the advantage of automatic execution. Thus it is possible to run a specific case with the given regulations to get a correct answer. An interesting aspect is the explanation of an answer, which is the reasoning chain of the response. Benefits include the user 's higher trust for the system. Formal methods also can be

used to detect anomalies, to investigate the effects of changes on the entire system and for debugging, which suggests changes to the regulations that will have as an effect the desired outcome.

The more recent approaches concluded that nonmonotonic systems with rules and priorities comprise appropriate solutions on the above requirements. Defeasible logic is the nonmonotonic reasoning approach that it is easily implementable and it has low computational complexity. It offers a natural way of representing regulations, by mapping them to rules. Since regulations may contradict one another, the sceptical behavior of defeasible logic prevents conflicting conclusions and none of the corresponding rules fires. Priorities among rules, based on several principles (recency, specificity, exception e.t.c), resolve these conflicts.

On the other hand, [3] showed that some features that appear to be useful or necessary for the analysis of regulations, require additions to defeasible reasoning and an improved underlying knowledge representation formalism. Our system captures these crucial aspects. It is based on a nonmonotonic reasoning formalism with improved expressiveness, an extension of defeasible logic with the required deontic notions of obligation, permission and prohibition. The system supports arithmetic and temporal operations, as the Prolog engines, used as reasoning engines, support useful built-in predicates. Finally, it supports the use of ontological knowledge by integrating and reasoning with RDF Schema ontologies and RDF data. Thus regulations make use of terminology and concepts, based on an ontological knowledge, which is relevant to an organization, which is a university in this use case.

Thus we describe a use case that shows how our system provides automated decision support for reasoning with university regulations. This a tool that would be used in many occasions by a student support service, a secretary, a lecturer etc. The system 's user interface provides functionalities for loading regulations as rules. It also uses a query editor that offers an easy way to formulate queries, that are processed successfully and responded.

6.2 University Ontological Knowledge

An RDFS ontology that models concepts and their relationships in the domain of the university is defined for this application and it is imported to the system. This document (available at the URL http://www.csd.uoc.gr/∼dimares/ontology/ University.rdfs) is represented as follow:

```
<?xml version="1.0" ?> <!DOCTYPE rdf:RDF [<!ENTITY xsd
"http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Student"> </rdfs:Class>
<rdfs:Class rdf:ID="Undergraduate">
```

```
    <rdfs:subClassOf rdf:resource="#Student"/></rdfs:Class>
<rdfs:Class rdf:ID="Postgraduate">
    <rdfs:subClassOf rdf:resource="#Student"/></rdfs:Class>
<rdf:Property rdf:ID="registrationYear">
   <rdfs:domain rdf:resource="#Student"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/></rdf:Property>
<rdf:Property rdf:ID="name">
   <rdfs:domain rdf:resource="#Student"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdfs:Class rdf:ID="Lecturer"></rdfs:Class>
<rdfs:Class rdf:ID="Professor">
    <rdfs:subClassOf rdf:resource="#Lecturer"/></rdfs:Class>
<rdfs:Class rdf:ID="AssociateProfessor">
    <rdfs:subClassOf rdf:resource="#Lecturer"/></rdfs:Class>
<rdfs:Class rdf:ID="AssistantProfessor">
    <rdfs:subClassOf rdf:resource="#Lecturer"/></rdfs:Class>
<rdfs:Class rdf:ID="Course"> </rdfs:Class>
<rdf:Property rdf:ID="code">
   <rdfs:domain rdf:resource="#Course"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="title">
   <rdfs:domain rdf:resource="#Course"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="units">
   <rdfs:domain rdf:resource="#Course"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property>
<rdf:Property rdf:ID="category">
   <rdfs:domain rdf:resource="#Course"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="enroll">
   <rdfs:domain rdf:resource="#Student"/>
   <rdfs:range rdf:resource="#ScheduledCourse"/>
    </rdf:Property>
<rdfs:Class rdf:ID="ScheduledCourse"> </rdfs:Class>
<rdf:Property rdf:ID="isGiven">
   <rdfs:domain rdf:resource="#ScheduledCourse"/>
   <rdfs:range rdf:resource="#Course"/> </rdf:Property>
<rdf:Property rdf:ID="semester">
   <rdfs:domain rdf:resource="#ScheduledCourse"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="academicYear">
   <rdfs:domain rdf:resource="#ScheduledCourse"/>
```

```
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property>
<rdf:Property rdf:ID="involves">
   <rdfs:domain rdf:resource="#ScheduledCourse"/>
   <rdfs:range rdf:resource="#Lecturer"/> </rdf:Property>
<rdf:Property rdf:ID="isTaughtBy">
   <rdfs:domain rdf:resource="#ScheduledCourse"/>
   <rdfs:range rdf:resource="#Lecturer"/>
   <rdfs:subPropertyOf rdf:resource="#involves"/>
    </rdf:Property>
<rdfs:Class rdf:ID="Exam"> </rdfs:Class>
<rdf:Property rdf:ID="examineCourse">
   <rdfs:domain rdf:resource="#Exam"/>
   <rdfs:range rdf:resource="#Course"/> </rdf:Property>
<rdf:Property rdf:ID="examineeStudent">
   <rdfs:domain rdf:resource="#Exam"/>
   <rdfs:range rdf:resource="#Student"/></rdf:Property>
<rdf:Property rdf:ID="period">
   <rdfs:domain rdf:resource="#Exam"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="examinationYear">
   <rdfs:domain rdf:resource="#Exam"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property>
<rdf:Property rdf:ID="grade">
   <rdfs:domain rdf:resource="#Exam"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property> <rdfs:Class
rdf:ID="BachelorThesis"> </rdfs:Class>
<rdf:Property rdf:ID="topic">
   <rdfs:domain rdf:resource="#BachelorThesis"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="hasAccepted">
   <rdfs:domain rdf:resource="#Undergraduate"/>
   <rdfs:range rdf:resource="#BachelorThesis"/>
    </rdf:Property>
<rdf:Property rdf:ID="submittedSemester">
   <rdfs:domain rdf:resource="#BachelorThesis"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#string"/> </rdf:Property>
<rdf:Property rdf:ID="submittedYear">
   <rdfs:domain rdf:resource="#BachelorThesis"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property>
<rdf:Property rdf:ID="thesisGrade">
   <rdfs:domain rdf:resource="#BachelorThesis"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
```

```
XMLSchema#integer"/> </rdf:Property>
<rdfs:Class rdf:ID="RecognizedCourse"> </rdfs:Class>
<rdf:Property rdf:ID="corresponding">
   <rdfs:domain rdf:resource="#RecognizedCourse"/>
   <rdfs:range rdf:resource="#Course"/> </rdf:Property>
<rdf:Property rdf:ID="passedBy">
   <rdfs:domain rdf:resource="#RecognizedCourse"/>
   <rdfs:range rdf:resource="#Student"/></rdf:Property>
<rdf:Property rdf:ID="passedGrade">
   <rdfs:domain rdf:resource="#RecognizedCourse"/>
   <rdfs:range rdf:resource="http://www.w3.org/2001/
XMLSchema#integer"/> </rdf:Property>

</rdf:RDF>
```

A graphical representation of the same RDF Schema is shown in 6.1.The above RDF Schema defines vocabulary, that describes a part of the domain of the University, which is related more to the undergraduate program of studies and the corresponding regulations that we model in this application. Thus we define the basic classes of *Course*, *ScheduledCourse*, *Exam*, *Student*, *Lecturer*, *BachelorThesis*, *RecognizedCourse*. The class Course defines all the lessons that comprise the program of studies of the department. The properties that apply to this class include all the basic information of the course like title, units, etc. ScheduledCourse is the class that defines the set of all courses that are taught or has been taught, during a semester. It contains information like the course which is taught, the semester, academic year etc. The class Exam defines the set of all examinations, and properties that apply to this class include the course that is examined, the examination period, the students that give this exam, their grades etc.

We also establish relationships between classes that define hierarchy. Thus we define the classes *Undergraduate* and *Postgraduate*, which are subclasses of class Student, and the classes *Professor*, *AssociateProfessor* and *AssistantProfessor*, which are subclasses of class Lecturer. Finally a hierarchical relationship is also defined for the property *isTaughtBy*, which is a subproperty of property *involves*.

This RDF Schema document is imported to the system and RDF translator transforms these concepts and relationships into the following Prolog rules:

```
fact(student(X))    :- fact(postgraduate(X)).
fact(student(X))    :- fact(undergraduate(X)).
fact(lecturer(X))   :- fact(professor(X)).
fact(lecturer(X))   :- fact(associateprofessor(X)).
fact(lecturer(X))   :- fact(assistantprofessor(X)).
fact(involves(X,Y)):- fact(isTaughtBy(X,Y)).
```

RDF documents that define instances regarding descriptions about students, courses, exams e.t.c are also imported and the terms which are used for the definition of the instances are references to this RDFS ontology. Parts of the initial instances are represented in the following RDF document (all the instances are avail-

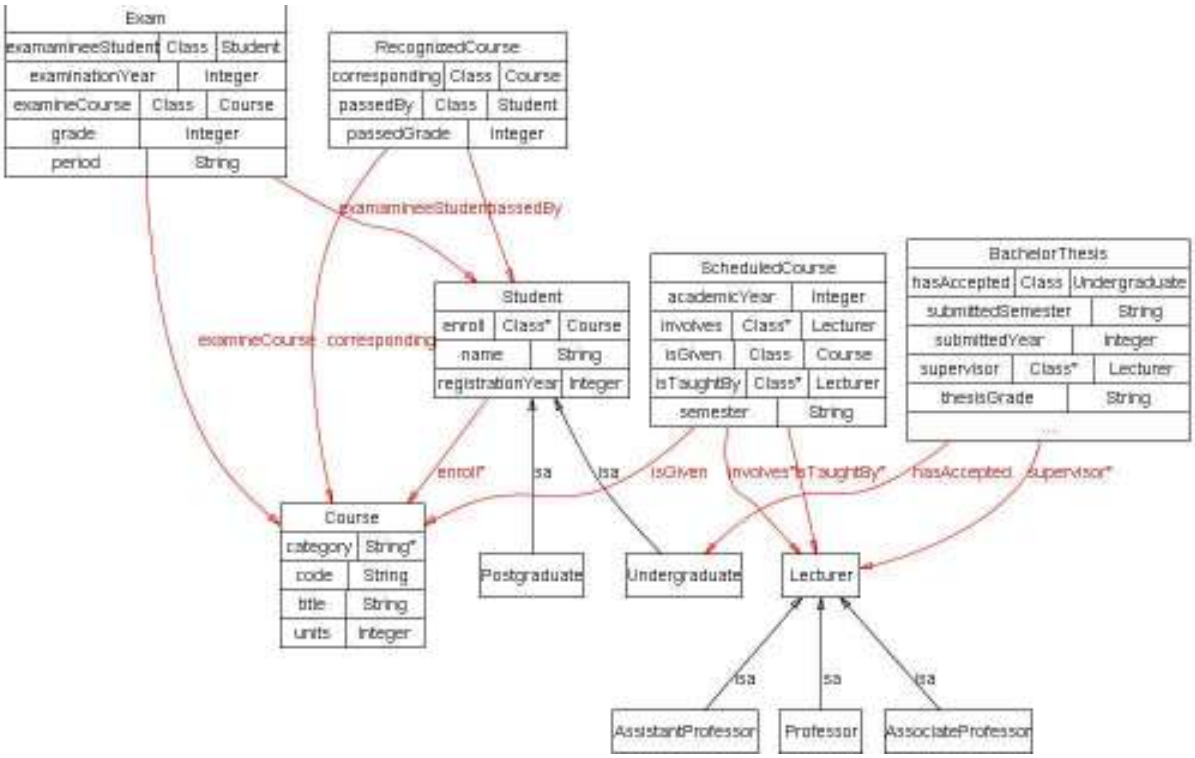

Figure 6.1: University RDF Schema

able at the URLs http://www.csd.uoc.gr/∼dimares/ontology/University_Data.rdf and http://www.csd.uoc.gr/∼dimares/ontology/Courses.rdf):

```
<?xml version="1.0"?> <!DOCTYPE rdf:RDF [<!ENTITY xsd
"http://www.w3.org/2001/XMLSchema#">]> <rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:uni=
"http://www.csd.uoc.gr/~dimares/ontology/University.rdfs#">

<uni:Course rdf:about="http://www.csd.uoc.gr/~hy280">
    <uni:code>cs-280</uni:code>
    <uni:title>Theory of Computation</uni:title>
    <uni:units>4</uni:units>
    <uni:category>core</uni:category>
</uni:Course>
<uni:Course rdf:about="http://www.csd.uoc.gr/~hy255">
      <uni:code>cs-255</uni:code>
      <uni:title>Software Technology Laboratory</uni:title>
      <uni:units>4</uni:units>
      <uni:category>core</uni:category>
</uni:Course>
<uni:Course rdf:about="http://www.csd.uoc.gr/~hy225">
```

```
      <uni:code>cs-225</uni:code>
      <uni:title>Computer Organization</uni:title>
      <uni:units>5</uni:units>
      <uni:category>core</uni:category>
</uni:Course>
<uni:ScheduledCourse rdf:ID="cs280fall2007">
      <uni:semester>fall</uni:semester>
      <uni:academicYear>2007</uni:academicYear>
      <uni:isGiven rdf:resource=
        "http://www.csd.uoc.gr/~hy280"/>
</uni:ScheduledCourse>
<uni:ScheduledCourse rdf:ID="cs255fall2007">
      <uni:semester>fall</uni:semester>
      <uni:academicYear>2007</uni:academicYear>
      <uni:isGiven rdf:resource=
        "http://www.csd.uoc.gr/~hy255"/>
</uni:ScheduledCourse>
<uni:ScheduledCourse rdf:ID="cs225fall2007">
      <uni:semester>fall</uni:semester>
      <uni:academicYear>2007</uni:academicYear>
      <uni:isGiven rdf:resource=
        "http://www.csd.uoc.gr/~hy225"/>
</uni:ScheduledCourse>
<uni:Undergraduate rdf:about="mailto:dimares@csd.uoc.gr">
      <uni:name>Nikos Dimaresis</uni:name>
      <uni:registrationYear>2006</uni:registrationYear>
      <uni:enroll rdf:resource="#cs100fall2007"/>
      <uni:enroll rdf:resource="#cs255fall2007"/>
      <uni:enroll rdf:resource="#cs225fall2007"/>
</uni:Undergraduate>
<uni:Exam rdf:ID="cs280january2008dimares">
      <uni:examineCourse rdf:resource=
        "http://www.csd.uoc.gr/~hy280"/>
      <uni:examineeStudent rdf:resource=
        "mailto:dimares@csd.uoc.gr"/>
      <uni:period>january</uni:period>
      <uni:examinationYear>2008</uni:examinationYear>
      <uni:grade>7</uni:grade>
</uni:Exam>
<uni:Exam rdf:ID="cs255january2008dimares">
      <uni:examineCourse rdf:resource=
        "http://www.csd.uoc.gr/~hy255"/>
      <uni:examineeStudent rdf:resource=
        "mailto:dimares@csd.uoc.gr"/>
      <uni:period>january</uni:period>
      <uni:examinationYear>2008</uni:examinationYear>
      <uni:grade>4</uni:grade>
</uni:Exam>
<uni:Exam rdf:ID="cs255september2008dimares">
```

```
      <uni:examineCourse rdf:resource=
        "http://www.csd.uoc.gr/~hy255"/>
      <uni:examineeStudent rdf:resource=
        "mailto:dimares@csd.uoc.gr"/>
      <uni:period>september</uni:period>
      <uni:examinationYear>2008</uni:examinationYear>
      <uni:grade>5</uni:grade>
</uni:Exam>
        ..............................
```

This is part of the RDF data that are loaded to the system. The RDF translator transforms these statements into Prolog facts, in the following form:

```
fact(course('http://www.csd.uoc.gr/~hy280')).
fact(code('http://www.csd.uoc.gr/~hy280','cs-280')).
fact(title('http://www.csd.uoc.gr/~hy280',
    'Theory of Computation')).
fact(units('http://www.csd.uoc.gr/~hy280',4)).
fact(category('http://www.csd.uoc.gr/~hy280',core)).
fact(scheduledcourse(cs280fall2007)).
fact(semester(cs280fall2007,fall)).
fact(academicyear(cs280fall2007,2007)).
fact(isgiven(cs280fall2007,'http://www.csd.uoc.gr/~hy280')).
fact(undergraduate('mailto:dimares@csd.uoc.gr')).
fact(name('mailto:dimares@csd.uoc.gr','Nikos Dimaresis')).
fact(registrationyear('mailto:dimares@csd.uoc.gr',2006)).
fact(enroll('mailto:dimares@csd.uoc.gr',cs280fall2007)).
fact(exam(cs280january2008dimares)).
fact(period(cs280january2008dimares,january)).
fact(examinationyear(cs280january2008dimares,2008)).
fact(grade(cs280january2008dimares,7)).
        ..............................
```

As we mentioned in section 5.4, the RDF translator represents RDF statements as triples and then transforms them into Prolog facts. That means that only binary predicates are loaded to the system. This results in writing rules with many binary predicates in their body and that makes more complicated the representation of regulations. For example, if the body of a rule requires information about a course with code *X*, like its units and category, this is represented as:

```
fact(course(X)),fact(code(X,Y)),fact(units(X,W)),
    fact(category(X,P)).
```

For reasons of readability and space in modelling regulations and in facilitating the formulation of queries, it is preferable to add the following rule, that defines a new predicate *course* with four arguments:

```
fact(course(Code,Title,Units,Category)):- fact(course(X)),
    fact(code(X,Code)),fact(title(X,Title)),
    fact(units(X,Units)),fact(category(X,Category)).
```

For the same reasons, the following clauses are defined:

```
fact(enroll(Student,Course,Semester,Year))  :-
    fact(enroll(X,A)),fact(student(X)),fact(name(X,Student)),
    fact(scheduledcourse(A)),fact(isgiven(A,B)),
    fact(code(B,Course)),fact(semester(A,Semester)),
    fact(academicyear(A,Year)).

fact(exam(Student,Course,Period,Year,Grade)):- fact(exam(A)),
    fact(examineestudent(A,B)),fact(name(B,Student)),
    fact(examinecourse(A,D)),fact(code(D,Course)),
    fact(period(A,Period)),fact(examinationyear(A,Year)),
    fact(grade(A,Grade)).

fact(recognized_passed_lesson(Student,Course,Grade)):-
    fact(recognizedcourse(A)),fact(corresponding(A,B)),
    fact(code(B,Course)),fact(passedby(A,D)),
    fact(name(D,Student)),fact(passedgrade(A,Grade)).

fact(accepted_thesis(Student,Semester,Year,Grade))  :-
    fact(hasaccepted(A,B)),fact(undergraduate(A)),
    fact(name(A,Student)),
    fact(submittedsemester(B,Semester)),
    fact(submittedyear(B,Year)),fact(thesisgrade(B,Grade)).
```

These clauses are loaded to the system as logic programs, before RDF data being loaded to the system, in order to be processed by these logical rules.

6.3 Modelling University Regulations

In this section we present how we model a variety of university regulations, taken from the undergraduate program of studies from the Computer Science department at the University of Crete. These are rules from the department 's policy in undergraduate studies and describe conditional entitlements. Thus we formalize them by using rules with the deontic mode of permission and obligation (representing indirectly prohibitions). A logic program is loaded to the system, that contains the logical rules of regulations (along with assistant clauses, like clauses for mathematical and temporal operations), and then the system is ready to response to user queries.

6.3.1 Enrollment in Courses

The following typical rule relates with the enrollment in courses for students:

A student has the permission to enroll in a course during a semester if he has passed the course's prerequisites, unless he has enrolled in courses this semester with total number of course units more than 35. A student is also forbidden to enroll in a course in a spring semester, a) if he has enrolled in the same course just the fall semester the same academic year (the previous year), or b) he has enrolled

in courses this academic year with total number of course units more than 65.

This is a typical rule with exceptions. In our logical framework of defeasible logic with extensions, these rules are represented with the use of defeasible rules:

$r1 : prerequisites\_passed(Student, Lesson, Semester, Year) \Rightarrow_{perm}$
$enroll(Student, Lesson, Semester, Year)$
$r2 : enroll(Student, Lesson, fall, Year) \Rightarrow_{obl}$
$\neg enroll(Student, Lesson, spring, Year + 1)$
$r3 : total\_semester\_units(Student, Semester, Year, Sum), Sum > 35 \Rightarrow_{obl}$
$\neg enroll(Student, Lesson, Semester, Year)$
$r4 : total\_semester\_units(Student, fall, Year, Sum1),$
$total\_semester\_units(Student, spring, Year + 1, Sum2),$
$add(Sum1, Sum2, Sum), Sum > 65 \Rightarrow_{obl}$
$\neg enroll(Student, Lesson, spring, Year)$
$r2 > r1, r3 > r1, r4 > r1$

The above defeasible theory is translated, according to the metaprogram approach, into the following clauses:

```
defeasible(r1,permission,
    enroll(Student,Lesson,Semester,Year),
    prerequisites_passed(Student,Lesson,Semester,Year)).
defeasible(r2,obligation,
    ~(enroll(Student,Lesson,spring,Year2)),
    [enroll(Student,Lesson,fall,Year1),inc(Year2,Year1)]).
defeasible(r3,obligation,
    ~(enroll(Student,Lesson,Semester,Year)),
    [total_semester_units(Student,Semester,Year,Sum),
    greater(X,35)]).
defeasible(r4,obligation,
    ~(enroll(Student,Lesson,spring,Year)),
    [total_semester_units(Student,fall,Year1,Sum1),
    inc(Year2,Year1),
    total_semester_units(Student,spring,Year2,Sum2),
    add(Sum1,Sum2,Sum),greater(Sum,65)]).
superior(r2,r1).
superior(r3,r1).
superior(r4,r1).
```

If the user issues a query about a student 's permission in enrolling a course during a semester, then the reasoning engine applies this query to the compiled logical programs. The body argument of the defeasible clauses contains predicates that may come from RDF/S documents, translated into facts, like the predicate *enroll(Student,Lesson,Semester,Year)*. Suppose that a query about the permission of student, with name *Nikos Dimaresis*, to enroll the course with code *cs-280* at spring semester of year 2008, is issued to the system. This query is expressed in Prolog syntax as *defeasibly(enroll('Nikos Dimaresis','cs-280',spring,2008),permission)*. If the information from the RDF document, that was showed in section 6.2, has

been loaded to the system, then *fact(enroll('mailto:dimares@csd.uoc.gr', cs280fall2007))* is loaded to the system. By using the processing rules, the reasoning engine also concludes that *fact(enroll('Nikos Dimaresis','cs-280',fall,2007))* is true. In this case, rule r2 is applicable and defeats rule r1. Thus the system responses by stating that the student does not have the permission to enroll again this course the next semester.

The body argument in defeasible clauses contains also predicates that are evaluated further in other clauses. An example is the predicate *prerequisites_passed(Student, Lesson,Semester,Year))* in rule *r1*, which checks if the student has passed the course's prerequisites until this semester. For example, the program of studies defines that a student is permitted to enroll the course with code *cs-255*, if he has passed the course with code *cs-150*. This is defined by the following clause:

```
strict(r114,knowledge,
    prerequisites_passed(X,'cs-255',Semester,Year1),
    [passed(X,'cs-150',Period,Year2,Grade),
    earlier_period(Semester,Year1,Period,Year2)]).
```

This is a strict rule for knowledge. Predicate *passed(X,'cs-150',Period,Year2,Grade)* is used in logic programs to describe the course that is passed by a student, along with the semester, the year and the grade. This is further evaluated in other more complicated clauses, by using arithmetic and list operations and ontological knowledge about the exams, translated into the facts of the form *exam(Student,Course, Period, Year,Grade)* etc. For reasons of space we will not refer further.

We also embed in the metaprogram the predicate *earlier_period(Period1,Year1, Period2,Year2)* in order to model temporal aspects for our application. This predicate compares two different moments. If the time defined by arguments *Period1,Year1* is more recent than the time defined by arguments *Period2,Year2*, then this literal is definitely provable in knowledge. An example is the following clause, which is embedded in the metaprogram:

```
strictly(earlier_period(fall,Year,january,Year),knowledge).
```

It defines that during a year, fall semester comes after month January. Thus these clauses facilitate the reasoning engine in evaluating queries correctly, taking in account temporal aspects.

Similarly predicate *total_semester_units(Student,Semester,Year,Sum)* calculates in variable *Sum*, the course units that has already been declared by the student in this semester. Finally the clauses contain predicates that offer arithmetic capabilities, like *add(Sum1,Sum2,Sum)*.

6.3.2 Exam Participation

The following typical rules relate with a student 's permission in giving exams during examining period and acquiring a grade:

A student has the permission to give an exam for a course in the examining period of January, if he enrolled in this course the fall semester, the same academic year. A Student has also the permission to give an exam in the examining period of June, if he enrolled in spring semester in the same year. Finally he can give an exam in September, if he enrolled in the course in fall or June semester, for the academic year that has passed.

These regulations are represented with the use of defeasible rules, as follow:

$r6 : enroll(Student, Lesson, fall, Year) \Rightarrow_{perm}$
$exam(Student, Lesson, january, Year + 1, Grade)$
$r7 : enroll(Student, Lesson, spring, Year) \Rightarrow_{perm}$
$exam(Student, Lesson, june, Year, Grade)$
$r8 : enroll(Student, Lesson, fall, Year) \Rightarrow_{perm}$
$exam(Student, Lesson, september, Year + 1, Grade)$
$r9 : enroll(Student, Lesson, spring, Year) \Rightarrow_{perm}$
$exam(Student, Lesson, september, Year, Grade)$

The above defeasible theory is translated, according to the metaprogram, into the following clauses:

```
defeasible(r6,permission,
    exam(Student,Lesson,january,Year,Grade),
    [enroll(Student,Lesson,fall,Year1),inc(Year,Year1)]).
defeasible(r7,permission,
    exam(Student,Lesson,june,Year,Grade),
    enroll(Student,Lesson,spring,Year)).
defeasible(r8,permission,
    exam(Student,Lesson,september,Year,Grade),
    [enroll(Student,Lesson,fall,Year1),inc(Year,Year1)]).
defeasible(r9,permission,
    exam(Student,Lesson,september,Year,Grade),
    enroll(Student,Lesson,spring,Year)).
```

Suppose again that we have loaded the RDF document from section 6.2, then again is concluded that *fact(enroll('Nikos Dimaresis','cs-280',fall,2007))* is true. Thus, if we query the system, this student has the permission to give an exam in course with code 'cs-280' at the examining periods of January and September in year 2008 (derived from applicable rules *r6* and *r8* respectively), but not at June in the same year (rule *r7* is not applicable).

6.3.3 Graduate Requirements

The following typical rules define when a student is permitted to graduate:

A student has the permission to graduate in a particular period, if he has fulfilled the following requirements: if he has passed a) all the core courses, b) at least two courses that belong to category e1, c) at least 28 units from courses that belong to categories e3-e9 and graduate courses, but at most three lessons at each category

and d) at least 158 units totally. On the other hand, it is forbidden to graduate if he has not been studying for at least 4 years in the department.

These regulations are represented with the use of defeasible rules, as follow:

$r10 : passed\_core\_lessons(Student, Period, Year),$
$\quad passed\_e1\_lessons(Student, Period, Year),$
$\quad total\_optional\_lessons\_units(Student, Period, Year, Sum1), Sum1 > 28,$
$\quad total\_units(Student, Period, Year, Sum2), Sum2 > 158 \Rightarrow_{perm}$
$\quad graduate(Student, Period, Year)$
$r11 : undergraduate(Student), registrationyear(Student, Year1),$
$\quad sub(Year2, Year1, X), X < 4 \Rightarrow_{obl}$
$\quad \neg graduate(Student, Period, Year2)$
$r11 > r10$

The above defeasible theory is translated, according to the metaprogram, into the following clauses:

```
defeasible(r10,permission,graduate(Student,Period,Year),
    [passed_core_lessons(Student,Period,Year),
     passed_e1_lessons(Student,Period,Year),
     total_optional_lessons_units(Student,Period,Year,Sum1),
     greaterEqual(Sum1,28),
     total_units(Student,Period,Year,Sum2),
     greaterEqual(Sum2,158)]).
defeasible(r11,obligation,~(graduate(Student,Period,Year2)),
    [undergraduate(Student),registrationyear(Student,Year1),
     sub(Year2,Year1,X),less(X,4)]).
superior(r11,r10).
```

These clauses use again predicates in their body, that are evaluated further in other clauses. These clauses calculate the course units of the optional lessons, completed by a student, and the course units of all the passed lessons. Other clauses check, by using list operations, if the student has passed all the core courses and if he has passed at least two courses that belong to *e1* category. Predicates that exist in the body argument are loaded as facts, like the predicates *undergraduate(Student)* and *registrationyear(Student,Year1)*, and arithmetic and comparison operations.

6.3.4 More Regulations for Enrollment

Bachelor thesis is represented as a core course with code cs-499. It is forbidden for a student to enroll in elaborating bachelor thesis, if he is not at least at the fifth semester of his studies, which means that he has been studying at least two years in the department. Undergraduate students may spend a work term for gaining experience, part-time or full-time, represented as optional courses with codes cs-499-3 and cs-499-6 respectively. A student is forbidden to enroll in a part-time work term, if he has already enrolled in courses this semester with more than 26 units. A student is also forbidden to enroll in a full-time work term, if has not completed at least 90 course units.

These regulations are represented with the use of defeasible rules, which conflict with rule r1, that was presented in subsection 6.3.1:

$r12 : undergraduate(Student), registrationyear(Student, Year1),$
$sub(Year2, Year1, X), X < 2 \Rightarrow_{obl}$
$\neg enroll(Student, 'cs-499', Semester, Year2)$
$r13 : undergraduate(Student),$
$total\_semester\_units(Student, Semester, Year, Sum), Sum > 26 \Rightarrow_{obl}$
$\neg enroll(Student, 'cs-499-3', Semester, Year)$
$r14 : undergraduate(Student), total\_units(Student, Period, Year, Sum),$
$Sum < 90 \Rightarrow_{obl}$
$\neg enroll(Student, 'cs-499-6', Semester, Year)$
$r12 > r1, r13 > r1, r14 > r1$

The above defeasible theory is translated, according to the metaprogram, into the following clauses:

```
defeasible(r12,obligation,
    ~(enroll(Student,'cs-499',Semester,Year1)),
    [undergraduate(Student),registrationyear(Student,Year2),
     sub(Year1,Year2,X),less(X,2)]).
defeasible(r13,obligation,
    ~(enroll(Student,'cs-499-3',Semester,Year)),
    [total_semester_units(Student,Semester, Year,Sum),
     greater(X,26)]).
defeasible(r14,obligation,
    ~(enroll(Student,'cs-499-6',Semester,Year)),
    [undergraduate(Student),
     total_units(Student,Period,Year,Sum),less(Sum,90)]).
superior(r12,r1).
superior(r13,r1).
superior(r14,r1).
```

Chapter 7

Conclusions and Future Work

In this report we argued that defeasible reasoning is a computationally efficient way of dealing with issues related to the modelling of policies and multi-agent systems. In particular, we have described how to enhance standard defeasible logic with agency, intention, permission, and obligation operators. Additionally, we outlined an implemented system that is also compatible with semantic web technologies.

At first, we reason why rule systems, especially the nonmonotonic ones, are expected to be part of the layered development of the Semantic Web. Then, we describe the logical formalism, on which the system is based on. We present defeasible logic and its main features, its declarative capabilities, and its low computational complexity. Then we explain why we extend modal and deontic operators, as a way to model two different aspects: the modelling of multi-agent systems and the modelling of policies.

We present the logic metaprogram, implemented in Prolog, that was used to implement the extension of defeasible logic and detail the implementation architecture of the system, its rationale, and its functionality. Finally, we present a use case that models a variety of university regulations, showing in practice the abilities and functionality of our system.

Our planned future work includes:

- Providing explanation mechanisms to help users in understanding policy decisions. When an answer is given, it is useful to provide a reasoning chain explaining the response. Such an explanation tool is meant to guide the user in acquiring the permissions necessary to get the desired services. Such features are essential in an open environment such as the Semantic Web, where the clients or users of a service are often occasional and do not know much about how to interact with the service.

- Supporting RuleML [76] syntax, the main standardization effort for rules on the Semantic Web. In [35], RuleML is extended by supporting modalities and defeasible logic as an inferential mechanism.

- Providing an experimental evaluation in order to measure the performance

of our system. Here, we have to deal with the problem that there are not any standard experimental tests for this formalism.

- Implementing load/upload functionality in conjunction with an RDF repository, such as RDF Suite [83] and Sesame [22]. This additional functionality will promote the integration of the system with the Semantic Web, and will make it more suitable for building web-based applications.

- Supporting integration with description logic based ontologies. The logical framework can be easily extended to ontologies which lie within the Horn expressible part of OWL, since these ontologies can be given a semantics based on a representation using Horn rules. In fact, DR-Prolog [2] supports reasoning with (parts of) OWL ontologies, through the transformation of many OWL constructs into rules.

- Providing applications of the developed implementation for modelling multi-agent systems.

Bibliography

[1] G. Antoniou and M. Arief. Executable declarative business rules and their use in electronic commerce. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 6–10, New York, NY, USA, 2002. ACM Press.

[2] G. Antoniou and A. Bikakis. DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):233–245, 2007.

[3] G. Antoniou, D. Billington, G. Governatori, and M. J. Maher. On the modeling and analysis of regulations. In *Proceedings of the Australian Conference Information Systems*, pages 20–29, 1999.

[4] G. Antoniou, D. Billington, G. Governatori, and M. J. Maher. A flexible framework for defeasible logics. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 405–410. AAAI Press / The MIT Press, 2000.

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

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

[7] G. Antoniou, D. Billington, and M. J. Maher. On the analysis of regulations using defeasible rules. In *HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 6*, page 6033, Washington, DC, USA, 1999. IEEE Computer Society.

[8] G. Antoniou and M. J. Maher. Embedding defeasible logic into logic programs. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, pages 393–404, London, UK, 2002. Springer-Verlag.

[9] G. Antoniou, M. J. Maher, and D. Billington. Defeasible Logic versus Logic Programming without Negation as Failure. *J. Log. Program.*, 42(1):47–57, 2000.

[10] R. Ashri, T. Payne, D. Marvin, M. Surridge, and S. Taylor. Towards a semantic web security infrastructure. In *In Proceedings of Semantic Web Services*, 2004.

[11] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[12] N. Bassiliades, G. Antoniou, and I. P. Vlahavas. DR-DEVICE: A Defeasible Logic System for the Semantic Web. In *PPSWR*, pages 134–148, 2004.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[14] D. Billington. Defeasible logic is stable. *J. Log. Comput.*, 3(4):379–400, 1993.

[15] G. Boella and L. van der Torre. Permissions and obligations in hierarchical normative systems. In *ICAIL '03: Proceedings of the 9th international conference on Artificial intelligence and law*, pages 109–118, New York, NY, USA, 2003. ACM Press.

[16] P. A. Bonatti and D. Olmedilla. Semantic Web Policies: where are we and what is still missing - ESWC'06 Tutorial, 2006.

[17] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[18] M. E. Bratman, D. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[19] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.0 (Second Edition). http://www.w3.org/TR/REC-xml-names/, 2006.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and o. Y. Franc. Extensible Markup Language (XML) 1.0 (fourth edition). Technical report, W3C, 2006.

[21] D. Brickley and R. Guha. Rdf vocabulary description language 1.0: Rdf schema, February 2004.

[22] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68, London, UK, 2002. Springer-Verlag.

[23] J. Broersen, M. Dastani, J. Hulstijn, Z. Huang, and L. van der Torre. The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 9–16, New York, NY, USA, 2001. ACM Press.

[24] J. Broersen, M. Dastani, and L. W. N. van der Torre. Resolving conflicts between beliefs, obligations, intentions, and desires. In *ECSQARU '01: Proceedings of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 568–579, London, UK, 2001. Springer-Verlag.

[25] J. Broersen, M. Dastani, and L. W. N. van der Torre. Bdioctl: Obligations and the specification of agent behavior. In *IJCAI*, pages 1389–1390, 2003.

[26] R. Conte. *Social Order in Multiagent Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[27] Cwm - a general-purpose data processor for the semantic web. http://www.w3.org/2000/10/swap/doc/cwm.html, 2007.

[28] M. Dastani and L. van der Torre. A classification of cognitive agents, 2002.

[29] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. dlvhex: A System for Integrating Multiple Semantics in an Answer-Set Programming Framework. In *WLP*, pages 206–210, 2006.

[30] D. Elgesem. The modal logic of agency. *Nordic Journal of Philosophical Logic*, 2(2):1–46, 1997.

[31] J. Gelati, A. Rotolo, G. Sartor, and G. Governatori. Normative autonomy and normative co-ordination: declarative power, representation, and mandate. *Artif. Intell. Law*, 12(1):53–81, 2004.

[32] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[33] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[34] T. Gordon. The pleadings game: An artificial intelligence model of procedural justice, 1993.

[35] G. Governatori. Representing business contracts in RuleML. *Int. J. Cooperative Inf. Syst.*, 14(2-3):181–216, 2005.

[36] G. Governatori, M. Dumas, A. H. M. ter Hofstede, and P. Oaks. A formal approach to legal negotiation. In *International Conference on Artificial Intelligence and Law*, pages 168–177, 2001.

[37] G. Governatori, M. J. Maher, G. Antoniou, and D. Billington. Argumentation semantics for defeasible logic. *J. Log. and Comput.*, 14(5):675–702, 2004.

[38] G. Governatori, V. Padmanabhan, and A. Sattar. A Defeasible Logic of Policy-Based Intention. In *AI '02: Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*, page 723, London, UK, 2002. Springer-Verlag.

[39] G. Governatori and A. Rotolo. A defeasible logic of institutional agency, 2003.

[40] G. Governatori and A. Rotolo. Defeasible Logic: Agency, Intention and Obligation. In *DEON*, pages 114–128, 2004.

[41] G. Governatori, A. Rotolo, and G. Sartor. Temporalised normative positions in defeasible logic. In *ICAIL '05: Proceedings of the 10th international conference on Artificial intelligence and law*, pages 25–34, New York, NY, USA, 2005. ACM Press.

[42] B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic, 2003.

[43] B. N. Grosof. Prioritized conflict handing for logic programs. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 197–211, Cambridge, MA, USA, 1997. MIT Press.

[44] B. N. Grosof. Representing E-Commerce Rules via Situated Courteous Logic Programs in RuleML. *Electronic Commerce Research and Applications*, 3(1):2–20, 2004.

[45] B. N. Grosof, M. D. Gandhe, and T. W. Finin. SweetJess: Translating DAML-RuleML to JESS. In *RuleML*, 2002.

[46] B. N. Grosof, Y. Labrou, and H. Y. Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 68–77, New York, NY, USA, 1999. ACM Press.

[47] I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 723–731, New York, NY, USA, 2004. ACM Press.

[48] I. Horrocks, Peter, H. Boley, Said, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml. available at: http://www.w3.org/Submission/SWRL/, May 2004.

[49] InterProlog - a Prolog-Java interface. http://www.declarativa.com/interprolog, 2007.

[50] Jena A Semantic Web Framework for Java. http://jena.sourceforge.net/index.html, 2007.

[51] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7 – 38, July 1998.

[52] O. Lassila and R. Swick. Resource description framework (rdf) model and syntax specification.

[53] A. Y. Levy and M.-C. Rousset. Combining horn rules and description logics in carin. *Artif. Intell.*, 104(1-2):165–209, 1998.

[54] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[55] M. J. Maher. A denotational semantics of defeasible logic. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 209–222, London, UK, 2000. Springer-Verlag.

[56] M. J. Maher. Propositional defeasible logic has linear complexity. *Theory Pract. Log. Program.*, 1(6):691–711, 2001.

[57] M. J. Maher. A model-theoretic semantics for defeasible logic, 2002.

[58] M. J. Maher and G. Governatori. A semantic decomposition of defeasible logics. In *AAAI/IAAI*, pages 299–305, 1999.

[59] M. J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001.

[60] W. Marek and M. Truszczynski. Nonmonotonic logic: Context-dependent reasoning. 1993.

[61] J. McCarthy. Epistemological problems of artificial intelligence. pages 46–52, 1987.

[62] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[63] Notation3 (N3) A readable RDF syntax. http://www.w3.org/DesignIssues/Notation3.html, 1998.

[64] D. Nute. Defeasible reasoning and decision support systems. *Decis. Support Syst.*, 4(1):97–110, 1988.

[65] D. Nute. *Handbook of logic in artificial intelligence and logic programming*, volume 3, chapter Defeasible logic. Oxford University Press, 1994.

[66] D. Nute. *Defeasible Deontic Logic*. Kluwer Academic Publishers, Dordrecht, 1997.

[67] D. Nute. Norms, priorities, and defeasibility. In H. Prakken and P. McNamara, editors, *Norms, Logics and Information Systems. New Studies in Deontic Logic and Computer Science*, pages 201–218. IOS Press, Amsterdam, 1998.

[68] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics, 2004.

[69] J. Pitt. *Open Agent Societies: Normative Specifications in Multi-Agent Systems*. John Wiley & Sons, 2004.

[70] H. Prakken and G. Sartor. A dialectical model of assessing conflicting arguments in legal reasoning. *Artificial Intelligence and Law*, 4(3-4):331–368, 1996.

[71] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[72] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[73] R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[74] R. Riveret, A. Rotolo, and G. Governatori. Interaction between normative systems and cognitive agents in temporal modal defeasible logic. In *Normative Multi-agent Systems*, 2007.

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

[76] RuleML. The RuleML Initiative website. http://www.ruleml.org/, 2007.

[77] G. Sartor. The structure of norm conditions and nonmonotonic reasoning in law. In *ICAIL*, pages 155–164, 1991.

[78] M. Sintek and S. Decker. Triple - a query, inference, and transformation language for the semantic web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, London, UK, 2002. Springer-Verlag.

[79] T. Skylogiannis, G. Antoniou, N. Bassiliades, and 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) on*

e-Technology, e-Commerce and e-Service, pages 44–49, Washington, DC, USA, 2005. IEEE Computer Society.

[80] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2007.

[81] S. Spreeuwenberg and R. Gerrits. Business Rules in the Semantic Web, Are There Any or Are They Different? In *Reasoning Web*, pages 152–163, 2006.

[82] SWI-Prolog. http://www.swi-prolog.org, 2007.

[83] The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web. http://139.91.183.30:9090/RDF, 2007.

[84] R. H. Thomason. Desires and defaults: A framework for planning with inferred goals. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 702–713, San Francisco, 2000. Morgan Kaufmann.

[85] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. Xml schema part 1: Structures. http://www.w3.org/TR/2000/WD-xmlschema-1-20000407/, 2000.

[86] A. von der Lieth Gardner. *An artificial intelligence approach to legal reasoning*. MIT Press, Cambridge, MA, USA, 1987.

[87] VRP - The ICS-FORTH Validating Rdf Parser. http://139.91.183.30:9090/RDF/VRP, 2007.

[88] G. Wagner. Web rules need two kinds of negation. In *PPSWR*, pages 33–50, 2003.

[89] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. HTTP://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1994.

[90] M. J. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, Massachusetts, 2000.

[91] XSB - Logic Programming and Deductive Database System for Unix and Windows. http://xsb.sourceforge.net, 2007.

[92] YAP Prolog. http://www.ncc.up.pt/ vsc/Yap, 2007.