ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

FACOLTÁ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA
in
FONDAMENTI DI INTELLIGENZA ARTIFICIALE LS

IMPLEMENTATION OF A NON-GROUND META-INTERPRETER FOR DEFEASIBLE LOGIC

CANDIDATO:
GIACOMO ACETO

RELATORE:
Chiar.mo Prof. Ing.
PAOLO TORRONI

CORRELATORI:
Prof. ANDREAS HAMFELT

Dr. JENNY ERIKSSON
LUNDSTRÖM

Anno Accademico 2009 – 2010
Sessione I

Dedicato ai
miei genitori

“Intus omne posui bonum;
non egere felicitate felicitas vestra est”
*Lucius Annaeus Seneca (*De Providentia*, Liber VI)*

Acknowledgement

This project would not have been possible without the support of many people. I wish to express my gratitude to my supervisors, Prof. Dr. Andreas Hamfelt and Dr. Jenny Eriksson Lundström who was abundantly helpful and offered invaluable assistance, support and guidance. A special thanks to my professor Paolo Torroni who gave me the opportunity to live this interesting experience and helped me immensely while writing this thesis.

Contents

Chapter 1

Introduction

In this chapter we provide a brief description of the thesis. In the first and second sections a complete introduction in both Italian and English languages is presented. Some answers to unsolved research questions are discussed in Section 3, finally, in Section 4 we provide a general, but comprehensive, description of the work in order to explain which are our goals and how we try to implement them.

1.1 Introduzione

Il ragionamento umano è un processo cognitivo affascinante e complesso che può essere applicato in diversi campi della ricerca come la filosofia, la psicologica, il diritto e la finanza. Purtroppo, sviluppare un software di supporto a tali aree che sia in grado di affrontare questo tipo di complessità risulta difficile e richiede un adeguato formalismo logico astratto.

In questa tesi intendiamo sviluppare un programma il cui compito sia quello di valutare una *theory* (un insieme di regole) in relazione ad un dato *Goal,* e restituire risultati del tipo: "*Questo Goal è derivabile a partire dalla KB (theory)*[1]". Al fine di raggiungere il nostro obiettivo dobbiamo analizzare diverse logiche e scegliere quella che meglio risponde ai nostri requisiti.

La logica, in genere, può essere vista come un tentativo di determinare se una data conclusione è logicamente implicata da un insieme di assunzioni T (theory). In realtà, quando ci

[1]Knowledge base. Nel seguito useremo l'abbreviazione KB per indicare l'insieme delle regole da usare per derivare un goal.

occupiamo della programmazione logica abbiamo bisogno di un algoritmo efficiente al fine di verificare tali implicazioni. In questo lavoro usiamo una logica piuttosto simile a quella umana. Infatti, il ragionamento umano richiede una estensione della logica del primo ordine capace di raggiungere una conclusione in base a premesse *non definitivamente vere*[2] che appartengono all'insieme della nostra conscenza. Dunque, il primo passo per lo sviluppo di tale software è l'implementazione di un framework basato su *defeasible logic*[3] in grado di *manipolare* e valutare le regole *defeasible* della nostra conoscenza.

Questo tipo di applicazioni risultano molto utili nell'ambito legale specie se dispongono di un framework argomentativo basato su una modellazione formale del gioco. In parole povere, poniamo che *theory* sia l'insieme delle leggi, che *keyclaim* sia la conclusione che uno dei partecipanti al dibattito vuole dimostrare (e che l'avversario vuole confutare), e dando la possibilità ai giocatori di poter inserire dinamicamente nuove regole nella conoscenza, allora, possiamo eseguire una competizione argomentativa tra le due parti e verificare se la conclusione raggiunta sia dimostrabile o meno a seconda della strategia di gioco usata dagli sfidanti.

Implementare un *game model* richiede un nuovo meta-interprete in grado di valutare il sottostante *defeasible logic framework;* infatti in accordo al teorema di Göedel (see on page 127), non è possibile valutare il significato di un linguaggio usando gli strumenti messi a disposizione dal linguaggio stesso, bensì, abbiamo bisogno di un meta-linguaggio capace di manipolare il linguaggio oggetto[4].

Quindi, piuttosto che un semplice meta-interprete, noi proponiamo un meta-livello contenente differenti meta-valutatori. Il primo di questi è stato descritto in precedenza, il secondo è necessario per implementare la competizione sul modello del gioco, un terzo meta-interprete sarà usato per cambiare le strategie di gioco e di esplorazione dell'albero di derivazione.

[2]Alcune regole possono essere usate anche se non sono sempre vere. Possiamo asserire che un uccello in generale vola anche se alcuni uccelli non ne sono in grado. Quindi, la regola non è innegabilemente vera, ma potrebbe essere accettata in un determinato contesto. (see Chapter 2)

[3]Defeasible logic è una logica non-monotona sviluppata da Nute per un efficiente applicazione del ragionamento defeasible. (see Chapter 2).

[4]Si noti che la valutazione di un linguaggio svolta nella maniera sbagliata, nel senso che viene effettuata usando gli stessi strumenti del lingiaggio stesso, conduce al paradosso.

1.2 Motivazioni

In questa sezione discuteremo le motivazioni alla base di questo progetto. Negli ultimi anni la ricerca su argomentazioni e ragionamento umano ha fatto passi da gigante e oggi nella letteratura scientifica è disponibile una elevato numero di pubblicazioni che spiegano diversi approcci per l'implementazion di un framework argomentativo o per la defeasible logic. Ciononostante, si avverte una notevole carenza di implementazioni, in particolar modo per l'interpretazione logica dei linguaggi, dal momento che i ricercatori hanno focalizzato il loro impegno su uno studio teorico delle possibile estensioni da integrare nei framework astratti già definiti. Tuttociò ha reso estremamente difficile i processo di testing relativo alle recenti evoluzioni di tali framework, e involontariamente ha comportato che lo studio di queste discipline ignorasse problemi relativi alla complessità computazione in termini di memoria occupata, carico computazionale del processore e tempo necessario per eseguire le applicazioni stesse.

Inoltre, potrebbero sorgere ulteriori problemi nel tentativo di sviluppare un software di supporto (come nel nostro caso). Noi vorremo implementare un framework argomentativo basato sul modello del gioco e sulla defeasible logic, ma, purtroppo, non ci sono DL-framework disponibili. Dunque, potrebbe essere interessante non solo provare a raggiungere il nostro obiettivo iniziale, bensì anche provare a implementare alcune estensioni pubblicate di recente relative alle *defeasible priority* e al *cambiamento dinamico della forza di un regola.*

In questo lavoro mostreremo come sia possibile implementare un meta-interpreter per defeasible-logic e un meta-livello in grado di eseguire la competizione tra due sfidanti; successivamente, analizzeremo la complessità risultante. Vogliamo dimostrare che il game-model è intrinsecamente problematico dal momento che coinvolge un elevato numero di strutture dati che devono essere allocate sullo stack del sistema comportando un notevole rallentamento dell'esecuzione. Quindi, alla luce delle conclusioni relative alle performance, analizzeremo sia come ottimizzare il codice sia la definizione teorica del game model, dal momento che potrebbero esserci delle imprecisioni che, trascurando casi particolari, non ci danno la possibilità di prevenire la costruzione dell'intero albero quando in realtà sarebbe possibile effettuare delle ottimizzazioni e ridurre la complessità. Nei capitoli 5 e 6 verrano presentati alcuni suggerimenti per lavori futuri. In più dimostreremo che una riduzione della complessità può essere attuata prevenendo che l'immissione di regole che portereb-

bero il sistema in stati instabili come ad esempio loop infiniti generati da paradossi. Dimostreremo che questo genere di problemi è risolvibile semplicemente spostandoci da un controllo di consistenza logica ad un controllo di consistenza dell'informazione, in altre parole, noi rigettiamo una regola se questa potrebbe essere inconsistente con la conoscenza comune, questo approccio comporta due benefici, in primo lugo riduciamo il numero di nodi nell'albero di derivazione, in più avendo eseguito un controllo a priori, non è più necessario effettuare un controllo sulla consistenza logica relativa a tali regole durante il controllo di derivabilità del keyclaim.

1.3 Architettura

Fin qui abbiamo introdotto le motivazioni del nostro lavoro ed abbiamo trascurato la descrizione del progetto al fine di mettere in luce quali sono le novità proposte e per quali motivi queste possono essere considerate rilevanti.

In primo luogo, dobbiamo creare un framework per la defeasible logic, l'implementazione di tale software sarà realizzata sfruttando le potenzialità dei linguaggi dichiarativi. Abbiamo usato un motore Prolog, in particolare Sicstus Prolog, e abbiamo testato l'applicazione su due macchine con i sistemi operativi Gnu/Linux e Windows. In più, abbiamo reso disponibile una interfaccia grafica basata sul linguaggio Java e su Rich Client Platform plugin messo a disposizione da Eclipse (Galileo). Nella figure 1.1 è possibile notare la struttura di base del nostro meta-interprete dove è enfatizzata la coesistenza di più meta-valutatori nello stesso livello, che chiameremo *meta-livello.*

In realtà, potremmo anche aggiungere diversi livelli per mostrare le relazioni tra alcuni meta-componenti, ma poichè tali componenti non sono sempre indipendenti e sono spesso incapaci di restituire un risultato se non coordinati da un softaware esterno o da un ulteriore meta-livello, è preferibile disegnare l'architettura definendo un meta-livello costituito da meta-componenti. Ciò che ci preme sottolineare è la struttura fortemente modulare. Poniamo che M1 sia il nostro meta-interprete per defeasible logic, e che M3 sia l'implementazione del meta-game-model; in questa architettura è possibile rimpiazzare un modulo di meta-livelli senza applicare ulteriori cambiamenti agli altri moduli. Per esempio potremmo sostiturire il criterio di gioco definito nel meta-livello con un nuovo criterio di gioco senza alterare il framework per defeasible logic sottostante. In questo modo possiamo garantire sia la modularità che la scalabilità del codice.

Il nostro obiettivo è l'implementazione dell'architettura che abbiamo appena descritto e sviluppare un framework argomentativo per lo svolgimento di un dibattito tra due giocatori. Ricordiamo che grazie a questo approccio modulare potremmo implementare differenti metodi di esecuzione del modello di gioco tra giocatori.

L'analisi implementativa metterà in luce una serie di problemi intrinsechi della formalizzazione del criterio di gioco, questo comporterà la definizione di due nuovi teoremi una proposizione e una serie di suggerimenti per l'ottimizzazione del codice per le future implementazioni del game model.

Per concludere suggeriremo anche delle possibili migliorie ed estensioni che si potrebbero integrare nel sistema quali Acid Cuts e Multiple Keyclaims.

1.4 Introduction

Human reasoning is a fascinating and complex cognitive process that can be applied in different research areas such as philosophy, psychology, laws and financial. Unfortunately, developing supporting software (to those different areas) able to cope such as complex reasoning it's difficult and requires a suitable logic abstract formalism.

In this thesis we aim to develop a program, that has the job to evaluate a theory (a set of rules) w.r.t. a *Goal,* and provide some results such as "*The Goal is derivable from the KB*[5] *(of the theory*)". In order to achieve this goal we need to analyse different logics and choose the one that best meets our needs.

In logic, usually, we try to determine if a given conclusion is logically implied by a set of assumptions T (theory). However, when we deal with programming logic we need an efficient algorithm in order to find such implications. In this work we use a logic rather similar to human logic. Indeed, human reasoning requires an extension of the first order logic able to reach a conclusion depending on *not definitely true*[6] premises belonging to a incomplete set of knowledge. Thus, we implemented a defeasible logic[7] framework able to manipulate defeasible rules.

[5]Knowledge base. In the following we will call KB the set of rules to be used to derive a Goal.

[6]Some rules can be used also if they are not always sure. We can assert that a bird *usually* flies,also if there are some birds that cannot flies. Thus, the rule is not undeniably true but it could be acceptable in some contexts. (see Chapter 2)

[7]Defeasible logic is a non-monotonic logic designed for efficient defeasible reasoning by Nute (see Chapter 2).

Those kind of applications are useful in laws area especially if they offer an implementation of an argumentation framework that provides a formal modelling of game. Roughly speaking, let the *theory* is the set of *laws*, a *keyclaim* is the conclusion that one of the party wants to *prove* (and the other one wants to *defeat*) and adding dynamic assertion of rules, namely, facts putted forward by the parties, then, we can play an argumentative challenge between two players and decide if the conclusion is provable or not depending on the different strategies performed by the players.

Implementing a game model requires one more meta-interpreter able to evaluate the *defeasible logic framework*; indeed, according to Göedel theorem (see on page 127), we cannot evaluate the meaning of a language using the tools provided by the language itself, but we need a meta-language able to manipulate the object language[8].

Thus, rather than a simple meta-interpreter, we propose a *Meta-level* containing different *Meta-evaluators*. The former has been explained above, the second one is needed to perform the game model, and the last one will be used to change game execution and tree derivation strategies.

1.5 Research question

In this section we discuss the motivations of our work. In the last years research on argumentation and human reasoning has made giant steps, nowadays in literature we can consult a lot of different approaches about the defeasible logic and argumentation framework. However, especially in defeasible logic interpretation, there is a deep lack of implementations. After the first Nute's implementation of DP-interpreter, the researchers committed themselves to add some extensions to this first framework just theoretically. Thus, it has become extremely difficult testing recently theoretical improvement, and evaluate their complexity such as memory or time requirements.

Furthermore, there may be problems attempting to develop supporting software. That is our case. We would like to implement an argumentation framework based on the game-model with defeasible logic, but there are no DL-framework available. So it could be interesting not only trying to achieve our initial goal, but also trying to test some extensions stated in the last years regarding *defeasible priority* and *dynamic changing of the strength*

[8]Note that evaluating a language in the wrong way, we mean using the language itself, will bring us to paradoxes.

of a rule.

In this work we will show how to implement the defeasible meta-interpreter and how to implement a meta-level able to perform a challenge between two players; later we will analyse complexity, in particular computational complexity. We want to demonstrate that game-model is intrinsically problematic since it involves an high number of different data structures that must be allocated on the stack and make the execution really slow. So we will survey both code programming optimisation and theoretical inaccuracies that could slow down the system since some general cases could be executed without building the whole tree; some suggestions to speed up system are presented as future work in Chapter 5 and 6. Furthermore, we will demonstrate that complexity reduction could be applied preventing the assertion of rules that would bring the system to unstable states such as paradoxical infinite loops. We will show that those problems are solved simply switching from logic inconsistency control to information inconsistency control, it means that we reject a rule if it could be inconsistent with the KB, consequently, we reduce the number of the nodes in tree derivation and we avoid the evaluation of a logical consistency check during the keyclaim evaluation.

1.6 Architecture

So far, we introduced the motivations of our work, but we left out a description about which are the novelties we want to propose and why they can be considered relevant.

First, we need to create a new framework for defeasible logic. We decided to implement the software using a declarative programming language such as Prolog; in particular we used Sicstus prolog, and we tested the application on GNU/Linux and Windows operating system. Furthermore, we realised a graphical interface using Rich Client Platform plug-in of Eclipse using Java programming language.

In figure 1.1, we show the logical architecture of the meta-interpreter. We emphasise the coexistence of multiple *meta-interpreters* in the same level.

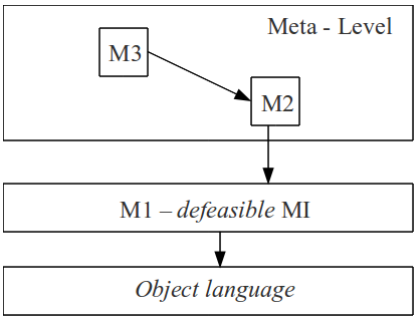


Figure 1.1: Defeasible Meta-Interpreter: logical architecture.

Actually, we could also add more levels in order to show the meta-relationship between meta-components, but, since those components aren't always independent and are usually unable to return a result unless they are coordinated by an external software or by a further meta-level, we prefer to draw the architecture defining a meta-level containing meta-components. It should be noted that this structure is strictly modular. Let M1 be our meta-interpreter and that M3 be the implementation of game-model; using this architecture is possible to replace a module without changing to other modules. For example, we could replace the game-model implementation with a new kind of game-model leaving the DL-framework without changes. In this way we guarantee modularity and scalability of the code.

Our aim is the implementation of the just presented architecture and develop an argumentation framework to perform the debate between two players. We remark that thanks to this modular approach, we could implement different kinds of game-models.

The analysis we carried out sheds light on some intrinsic problem of the formal game-model, thus, we refine some conclusion about the model, introducing two theorems, one proposition and some suggestions for code optimisation and for future implementations of this game-model.

Finally, we present some interesting topics for future works that could be integrated into our system such as: *Acid Cuts* and *Multiple Keyclaim.*

1.7 Outline

In the next chapters we present a brief introduction to non-monotonic reasoning (Chapter 2) and we describe defeasible reasoning. In Chapter 3 we provide a complete analysis on the

argumentation framework, the relation between semantics and extensions and we introduce an approach to perform a game-model debate. In Chapter 4 we discuss the meta-languages, their definition and their representation, we shows differences between ground and non-ground meta-interpreters and the motivations of using a non-ground meta-interpreter in this work. In Chapter 5 we describe clearly the milestones of the implementation of the project[9]. In Chapter 6 we will realise step by step a case of study in order to prove the correctness of the software and to check the complexity in terms of memory and CPU's consume and we conclude showing which are the future applications.

[9] A complete explanation can be found on http://splogad.altervista.org

Chapter 2

Non Monotonic Reasoning

In this chapter we introduce non-monotonic reasoning and different kinds of logic such as default logic and we discuss if they are applicable for our purpose. As we said in the previous chapter, we need a non-monotonic formalism as basic abstract starting point for our framework implementation. However, even starting from now, we need to survey in parallel theoretic assumption and their application on a concrete logic language (i.e. prolog) since it's needed to evaluated feasibility of our approaches.

We want to focus on the knowledge representation and knowledge assimilation. The first one implies an accurate analysis about the knowledge formalization, the second one, in literature, is usually associated to the problem of the database update[1], namely, if adding a new sentence in the database could alter the old sentences or if the new sentence is inconsistent with the other ones.

Please note that in monotonic logic the addition of a new sentence in a knowledge repository doesn't alter previous knowledge. This feature represents a strong limitation as we cannot emulate human reasoning in which knowledge usually improves over the years also refuting previous not provable beliefs (before accepted as if they were absolutely true).

2.1 Non-monotonic reasoning

We say that an inference is non-monotonic if it is defeasible; it means that the basics of a knowledge repository sometimes could generalize and seem true until we demonstrate that they are wrong. Human beings adopt the same behavior, thus, while discovering new

information are inclined to change idea, namely, withdraw the inference that before was considered undoubtedly true.

Why are humans often wrong? It was due to generalize some facts as if they were always true, so when we deal with a particular exception we discover that our knowledge was not sufficient.

The typical example about non-monotonic inference in literature is the bird example. If we know that *Tweety* is a bird, then we can say that *Tweety* can fly (as our knowledge suggests that usually birds fly). But we should withdraw our conclusion as soon as we know that *Tweety* is not a *typical* bird, it is a penguin. Thus, lack of informations can lead us to commit mistakes in our conclusions. It means that adding new informations, in this case *Tweety* is *atypical*, our conclusions could change.

Normally, in monotonic logic, we could not conclude that *Tweety* can fly just knowing that it is a bird, that a rule asserts that usually birds fly and that there are some extensions (i.e. penguins) that could state that *Tweety* is an atypical bird. In the example above, we introduced a new topic, we mean, *how to deal with incomplete knowledge?* The *Tweety* example sheds light on the advantage of using an improved knowledge system that is able to reach ambiguous conclusions.

The so-called ***frame problem*** is one of the most interesting case in which we can apply a non-monotonic reasoning. The *frame problem* is the following one:

> *"In order to plan how to reach its goals, an artificial agent will need to know what will and what will not change as a result of each action it might perform."*

The problem of listing all the things that will not change after performing an action could be impracticable. The solution is to use non-monotonic reasoning. With this new approach we assume that *usually* no actions alter the system, except some known actions. Thus, we assume a generalized idea about our actions in order to list the ones who change our environment, rather than the actions who leave the system unchanged.

One more non-monotonic application is known as ***closed world assumption***; by using a particular form of negation (*see later*) it operates assuming that repository *(usually*) contains all the relevant information. Let S be a sentence that cannot be proved by the knowledge in the repository KB and that adding the sentence t in the KB, S is proved, thus the same repository with a new sentence can derive or not a given sentence. That is the typical

behavior of a non-monotonic system. We use this occasion to introduce *Negation as Failure* (see on page 22), namely, a particular form of negation widely used in logic programming.

Let the following program $\mathcal{P}$[2]:

```
car(volvo).

car(saab).

car(ferrari).

driver(giacomo).
```

We are sure that the goal *car(giacomo)* is non-derivable from the program $\mathcal{P}$, and intuitively we would answer *car(giacomo)* = *false.* In other words, let $\mathcal{A}$ be a *gound*-term :

$CWA(\mathcal{P}) = \{ \sim\mathcal{A} \mid$does not exist a refutation SLD for $\mathcal{P} \bigcup \{\mathcal{A}\}\}$

Example:

```
capital(stockholm).

city(X):- capital(X).

city(uppsala).
```

And the following query:

```
?- not(capital(uppsala)).

true.

?- not(capital(bologna)).

true.
```

Thus, our program just knowing that Stockholm is the capital is able to answer if whatever city is a capital. It appears "*clever*" as it says that Uppsala is not a capital although it does not contain informations such as *"not(capital(uppsala))*". But it isn't intelligent. What happens if I ask "*capital(rome)*"?

Dealing with incomplete knowledge and defeasible reasoning we have to strike a balance between which are the contexts in which is the case that the system is able to give an answer and what it really knows.

2.2 Finding a non-monotonic formalism

2.2.1 Default logic

In order to formalize a non-monotonic system, we need to add new rules to the *first-order logic.* This work has been performed by Raymond Reiter, who defined the ***default logic*** adding the *default rule*:

$$\frac{p : q}{r} \tag{2.1}$$

In this rule *p* identifies the *prerequisite*, *q* is the *justification*[1] and *r* is the *consequent* and it means that if *p* is true and assuming *q* is consistent with the rest of our *KB*, then we conclude *r*. In the simpler cases *q* and *r* can coincide, i.e. the bird example:

$$\frac{bird(X) : flies(X)}{flies(X)} \tag{2.2}$$

Thus, the sentence *"assuming flies(X) is consistent with the rest of our KB,"* could not be verified if in our *KB* there are some information about *atypical* birds. This could seem a trivial case, but it becomes more complex assuming that *consistency* depends not only on the initial data but also on the various combinations of *consequences* that derive from them. Reiter also introduced the *extensions.* He defined an *extension* for a *default theory* in the following way:

> Let $\mathcal{T}$ is a *default theory* composed by:
>
> - a set $\mathcal{W}$ of premises;
> - a set $\mathcal{D}$ of rules;
>
> an *extension* is a set of sentences $\mathcal{E}$ that can be derived from $\mathcal{W}$ applying as many of the rules of $\mathcal{D}$ as possible ***without generating inconsistency.***

Default logic definition is important for our purpose as it can generalize some conclusions without including in the knowledge all the rules needed for their derivation. Let us suppose

[1] We can put one or more justifications.

that two friends are walking in a park and one of them, say Jim, sees a dog behind a bush and that they have never seen that dog before. Now, Jim asks to Paul (the other guy) "*How many paws does that dog have?*". Paul could say "*it should have four paws*", in other words he says that *usually* or *by default* a dog has four paws, but it is not undoubtedly true. The only way to know if that dog has four paws is to wait until the dog gets out that bush and see. In this book we will not reveal the end of this skit in order to better impress on the reader the lack of absolute certainty of this approach.

2.2.2 Auto-epistemic logic

Another kind of logic whose definition is close to the default one is the *auto-epistemic logic* (from greek "$\varepsilon\pi\iota\sigma\tau\eta\mu\eta$" - "knowledge, science"). Whit this logic we can infer somethings about the world just considering my own knowledge and using my own way of think. The typical example is:

> *"From the fact that I do not believe that I owe you a million of pounds, I can infer that I do not owe you a million pounds, since I would surely know if I did."*

So coming back to the bird example is like to say that if someone cannot believe that *Tweety* can't fly, then *Tweety* flies. *Auto-epistemic logic* syntax extends that one of propositional logic with a modal[2] operator $\Box$ and its meaning is the following:

let $\mathcal{F}$ be a Formula.

[2] A brief introduction to modal logic is needed:

Modal logic is a type of formal logic that extends the standards of formal logic to include the elements of modality (*probability, possibility* and *necessity*).

Modals qualify the truth of a judgment. For example, if it is true that "*John is happy*", we might qualify this statement by saying that "*John is very happy*", in which case the term "*very*" would be a modality. Traditionally, there are three "*modes*" represented by modal logic, namely, `possibility`, `probability`, and `necessity`.

The basic unary (1-place) modal operators are usually written $\Box$(= *"it is believed that..."*) for Necessarily and $\diamond$ for Possibly.

In a classical modal logic, each can be expressed by the other and negation:

- $\diamond$P$\leftrightarrow \neg \Box \neg$ P;
- $\Box$ P $\leftrightarrow \neg \diamond \neg$ P.

Thus it is possible that it will rain today if and only if it is not necessary that it will not rain today.[3]

- $\Box\ \mathcal{F}$ means that $\mathcal{F}$ is known;
- $\Box\neg\ \mathcal{F}$ means that $\neg\ \mathcal{F}$ is known;
- $\neg\Box\ \mathcal{F}$ means that $\mathcal{F}$ is not known.

In our case as $\Box\neg$ ($\neg$*flies(Tweety)*), then Tweety can fly. This kind of logic, instead of being used as a formal model for our framework could be used for writing a theory. A new example could be: "*we can assert that birds fly because nobody has seen a penguin before, and so we cannot prove neither that penguins exist nor they don't fly*"; in this case we extended the auto-epistemic logic from personal reasoning to community beliefs. Let's go back to the past for a moment, in middle age for example. Now, we take into consideration a small village mainly inhabited by peasants where nobody of them has ever seen a penguin before. Defeasible inference usage mixed with *auto-epistemic/community-epistemic* logic would not cause logic damages in their daily life, and this example represents a context in which assuming some defeasible facts doesn't seriously affect defeasible reasoning.

2.2.3 Circumscription

One more way to formalize non-monotonic reasoning is Circumscription that circumscribes some rules of thumb using abnormality predicates, in order to apply the rules to only those things to which they must apply.

In other words, rather than assuming some general rules, we limit the target items to which to apply our rules, just improving them.

$$\text{“}\forall X(Bird(X)\&\neg Abnormal(X)) \rightarrow Flies(X)\text{”}$$

In our case we have to explicitly say if X is abnormal or not.

We leave this topic aside for a while, as we will come back later with a deeper analysis.

All the models that we introduced have their strengths and weakness, but it is important to underline an interesting aspect; every model defines different *extensions*[3] starting by the same premises. The following Nixon example shows how can be possible to generate some incompatible *extensions* (which is obviously a side effect).

Starting by the premises:

- *Nixon is a Quaker;*

[3] An comprehensive definition of extensions will be provided in the next chapter.

- *Nixon is a Republican.*

and according to the following rules:

- *Quakers are typically pacifists;*
- *Republicans are typically not pacifists.*

Each of these rules block the application of the other one. Having multiple extensions is not a weakness, it is obvious that starting from premises that do not describe the whole world we get more interpretations. Each extension is a particular way of reasoning on the premises. Since now we can define two different strategies.

Let $\mathcal{E}$ be a set of extensions:

credulous strategy if we simply believe to one of the extension in $\mathcal{E}$;

skeptical strategy if we believe on those claims that appear in every extensions in $\mathcal{E}$;

Roughly speaking, an extension contains all the *argument* that can *stand together*, a skeptical extension is the intersection of all credulous semantics (semantics will be discussed in detail 3.4).

Nevertheless, a plausible theory can generate an unacceptable extension (Take a look at the *Yale shooting problem* below).

Finally, we would like to spend some words on *probabilistic logic.* It provides some threshold and a value (typically a value that varies over time) that is assigned to a possible *conclusion*. When this value exceeds the threshold, the related conclusion can be taken as plausible. It is non-monotonic since the addition of premises can modify the threshold values or the values assigned to the prepositions.

2.3 Problems dealing with computational logic

In this section we will concretely explain, as described in [4], how to represent non monotonic reasoning examples seen before using logic programming. We provide this deepening as it will be essential to understand the prosecution of this work.

We take for granted that the reader knows differences among *abduction, deduction* and *induction,* nevertheless we provide a simple example from[5] to better understand those kinds of reasoning. Let define a reasoning composed by a *rule* a *case* and a *result*:

Deduction

- Rule: *All the beans in that bag are white.*
- Case: *Those beans come from that bag.*
- Result: *Those beans are white.*

Induction

- Case: *Those beans come from that bag.*
- Result: *Those beans are white.*
- Rule: *All the beans in that bag are white.*

Abduction

- Rule: *All the beans in that bag are white.*
- Result: *Those beans are white.*
- Case: *Those beans come from that bag.*

Those approaches have different behaviors since a*bduction* and *deduction* are the best way to formalize (in programming logic) predicates able to return results / cases from rules, instead, using *induction* we get rules starting from cases / results and it becomes interesting during the evaluation of a goal and also executing a meta-level able to manipulate a rule.

2.3.1 Abduction and integrity constraints

Abduction is a human reasoning behavior that can be successfully applied in computational logic to solve some of the problems discussed in literature, such as *Database updates problem*, and it provides generalization for *negation by failure.*

An abductive explanation of a conclusion $\mathcal{C}$ is a set of sentences called $\triangle$ such that:

1. $\mathcal{T} \bigcup \Delta$ logically implies $\mathcal{C}$;
2. $\mathcal{T} \bigcup \Delta$ satisfies $\mathcal{I}$;

Where $\mathcal{T}$ is a theory containing both rules and hypothesis, $\mathcal{I}$ is a set of integrity constraints.

From now on we consider that constraints are formalized as denials and interpreted by a logic with a semantic for *negation by failure.* To better understand how to deal with constraints with abduction a brief deepening on this kind of reasoning is needed. Abduction is the process of inference that produces a hypothesis as its end result [6]. Commonly, the term abduction is presumed to mean the same thing as hypothesis, that is due to the fact that the conclusion depends on a *strong* premise that is for sure and a *weaker* premise that is uncertain, or better, is certain only in given domains. Thus, abduction involves weak premises that we cal *hypothesis* and some constraints that restrict the degrees of freedom (possible values) [7].

Kowalski in [4] points that if transform an abductive program into a logic theory T' we can get, as we show below using a *Horn-clause-free* theory, the same results by deduction.

A typical application of *Abduction* explanation is fault diagnosis. This example shows the possible causes of a woobly-wheel. In this case we notice that there are two possible options, tyre is flat or spokes are broken, but adding a constraint such as tyre holds air, then the first cause should be rejected. Given the following theory and Constraint:

Program

```
wobbly-wheel ←flat-tyre                    (1)
wobbly-wheel ←broken-spokes                (2)
flat-tyre ← punctured-tube                 (3)
flat-tyre ← leaky-valve                    (4)
```

Constraint

```
← flat-tyre, ¬tyre-holds-air
```

Hypothesis the following sub goals can be considered hypothesis if they are consistent with KB and integrity constraint

```
punctured-tube
leaky_valve
broken_spokes
```

In order to know which is the cause of the failure, we try to get the *abductive explanation* of the goal (query):

? *wobbly-wheel*[4]

As we can see all assumptions are consistent with the theory. But adding a new information as:

tyre-holds-air

(3),(4) bring the system to be inconsistent with the constraint and so we should withdraw non-monotonically *punctured-tube* and *leaky-valve.* In other words, if we are sure that tyre holds air, is not possible that the tube or the valve is broken.

Furthermore, we can also perform fault diagnosis using *dialectic* reasoning. In this case we have to reverse cause and effect relations and ignore the constraints. To apply dialectic reasoning we have two ways: logic programming or non-Horn clauses. The first option may have the following representation:

```
possible(flat-tyre) ← possible(wobbly-wheel)
possible(broken-spokes) ← possible(wobbly-wheel)
possible(punctured-tube) ← possible(flat-tyre)
possible(leaky-valve) ← possible(flat-tyre)
possible(wobbly-wheel)
?- possible(X)
```

This is a lower-level representation and predictable. Programming representation it's really similar to programming languages and, usually, provides an easy way to write a program. Nevertheless, there is also a side effect in term of expressivity.

The other way is non-Horn representation:

```
flat-tyre ∨ broken-spokes ← wobbly-wheel
punctured-tube ∨ leaky-valve ←flat-tyre
wobbly-wheel
```

[4]Using this goal, is the same that using all the sub goals and it is a backward-reasoning.

Since the program is *Horn-clause-free,* we can list all possible failure causes or represent the integrity constraints. For example, if we add some information as:

```
← flat-tyre, ¬tyre-holds-air
tyre-holds-air ∨ flat-tyre ∨ broken-spokes ← wobbly-wheel
punctured-tube ∨ leaky-valve ←flat-tyre
wobbly-wheel
```

we notice that it is possible to derive monotonically the conclusion:

```
broken-spokes
```

In this simple example we applied a particular approach called "*abduction through deduction*".

2.3.2 Default reasoning[5]

Abduction provides a natural mechanism for performing reasoning [8]. Reiter has illustrated the *Nixon example*[6] [9] that it is useful to introduce the *conflicting defaults.* This example known also as *Nixon Diamond* is the typical case is a scenario in which default assumptions can lead to mutually inconsistent conclusions. We know that usually quakers are pacifist and that republican usually are not pacifist. What happen if we say that someone, say, Nixon is both a quaker and a pacifist? We could reach both conclusion pacifist and its negation.

Using the abduction representation the example might take the form:

Theory

```
pacifist(X) ← quaker(X), normal-quaker(X)
hawk(X) ← republican(X), normal-republican(X)
quaker(nixon)
republican(nixon)
```

Constraint

Hypothesis	Conclusion
normal-quaker(nixon)	pacifist(nixon)
normal-republican(nixon)	hawk(nixon)

Table 2.1: Hypothesis and relative conclusion in Nixon Diamond Example with Abduction approach.

```
← pacifist(X), ¬ hawk(X)
```

According with table 2.1 we can infer both the conclusions with the appropriate hypothesis but not at the same time. This is the typical case where two hypothesis are both consistent with the constraints, but together are inconsistent, namely, this is a *conflicting defaults* phenomenon. In other words, we would like to represent an inconsistency between two hypothesis that are not involved in a constraint. In this case we should add a new predicate able to notify to the system that it is not allowed to assert both conclusions also if the abducible rules are consistent. So, we developed *incompatibility/2* predicate that represents the above constraint (see on page 65).

2.3.3 Negation as Failure (NaF)

Negation as failure is an attempt (rather similar to CWA) to obtain answer by a limited Knowledge repository. NaF derive negations of atoms whose proof finishes with a failure in finite time. Let $\mathcal{P}$ is a program, the set FF($\mathcal{P}$) contains the atoms[7] $\mathcal{A}$ such that their proof fails in a finite time. To better understand Negation as Failure in Prolog we recommend to consult the deepening on page 129.

What we want to focus is that abduction simulates and generalizes negation as failure.

If $\mathcal{P}$ is the following:

```
p ←not q
q ← r
```

to apply an abductive reasoning we need two more element: an hypothesis and one or more constraints.

[5] See *Default logic* on page 14

[6] We saw this example on page 16

[7] Remember that prolog does not apply a safe rule to select elements to inspect. It takes the leftmost literal without checking if it is *ground* or not.

The hypothesis could be the following:

"is the case to assume $\neg q$ to hold, only if $\neg q$ is consistent"

the constraint state that q cannot be true and false at the same time:

```
←q, ¬(¬q)
```

Using backward reasoning and integrity constraints, to infer that *p* is true, we should check *not q* for consistency, if it fails, it means that *not q* is consistent and the result is true. In a more complex case we could have more constraints and, mostly, nested negation :

```
pacifist(X) ←not republican(X)
republican(X)← not democratic(X)
```

And two constraints:

```
← republican(X), not republican(X)
← democratic(X), not democratic(X)
```

Checking not republican(X) for consistency we have two possible options:

1. not republican(X) is inconsistent **with** the hypothesis not democratic(X). So pacificst(X) fails.

2. not republican(X) is consistent **without** the hypothesis not democratic(X). So pacificst(X) succeeds.

Now the problem is that point 1 can be applied using NaF, instead, point 2 not. To eliminate 2. we can add a new constraint:

```
not democratic(X) ∨ democratic(X)
```

Kawasaki in [4] state that we can see this relation as meta-level or modal epistemic statement [10] that means:

either `democratic(X)` *or* `not democratic(X)` *can be derived from the theory.*

In this way if *democratic(X)* is not proved, we assume the hypothesis *not democratic(X).* The atom *democratic(X)* is not proved since when we deal with abduction just to simulate NaF, **only negative atoms are assumed as hypothesis**[8]**.**

Leaving aside particular cases we saw above[9] in this chapter we would like to focus on different kind of approach to build a defeasible logic, but also different ways to reasoning in order to get you used to those concepts that will be used later for the implementation.

2.3.4 The Yale Shooting Problem

The Yale Shooting Problem is one of the most discussed in the literature. Indeed, applying some minimal model such as circumscription and default logic, we get unacceptable results, instead, NaF through abduction gives correct result.

The problem is the following. We want to determine if after shooting a turkey, called Fred, remains alive or it dies:

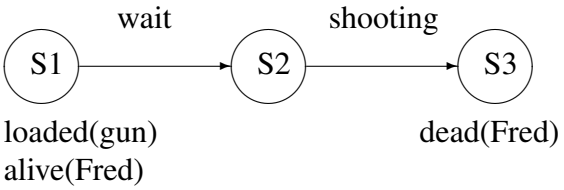


Figure 2.1: Yale shooting problem

Neither *circumscription,* nor other minimizing approach can solve this problem as they accept unpredictable solutions. Minimizing the changes, the expected solution allows two condition changes: the first one to load the gun, the second one to transit the turkey's state from alive to dead (or not(alive)).

Let suppose to represent the problem using the graph in Figure 2.1. It should be noted that this graph requires a *temporal analysis* since it involves change of status (*effect*) caused by *events.* Given a notation, introduced by Hanks and McDermott in [11], *t(P,S)* means that property P holds in situation S. Furthermore, we need two more representations: the first one to explain the result caused by the occurrence of an event in a given situation *result(E,S)*

[8] A deep explanation about how to simulate NaF using Abduction is discuss in [4] *(on page 10)*

[9] We will continue to use most of them throughout the book to easily understand the final behavior of the application.

and the other one to represent a case of abnormality *ab(P,E,S)* that means that property P is abnormal w.r.t. the occurrence of event E in situation S.

Given the following *program*:

```
1 t(alive, s0)
2 t(loaded, result(load, S))
3 t(dead, result(shoot))  t(loaded,S)
4 t(P, result(E,S)) t((P,S), not ab(P,E,S)
5 ab(alive, shoot, S) t(loaded,S)
```

in line 1 we define the property Fred is alive at the beginning, in line 2 we define that after we load the gun we get the property (loaded), in line 3 we say that if the gun is loaded and after we shoot, Fred dies. In line 4 there is a control in order to avoid that the result involves a property that is abnormal in a give situation S. Last line contain the typical abnormality statement: if we shoot with a loaded gun, Fred cannot remain alive.

Now, let suppose we want to apply *circumscription,* minimizing the predicate ab. Intuitively, the control of line 4 will fail passing P = alive, since the last line has been removed. It means that our system is able to reach both conclusions: Fred alive and Fred died after shooting with a loaded gun.

Dealing with *frame problem*[10] approach we would accept the following solution: "that after shooting, system could pass to unloaded gun state; leaving Fred *alive*. Indeed, adding the following rule:

t(unloaded, result(shoot)) $\leftarrow$ t(loaded,S)

and after shooting, system could say that Fred is alive and gun unloaded[11].

What we tried to show in this section, is that sometimes *default logics* and *minimizing approaches* such as *circumscription* and *frame problem*, are not able to get the right solution since in those formulations *"is missing [...] an appropriate analogue of the disjunctive integrity constraints"[4].*

[10]The frame problem was initially formulated as the problem of expressing a dynamical domain in logic without explicitly specifying which conditions are not affected by an action, nowadays, this problem is known as the problem of limiting the beliefs that have to be updated in response to actions. In other words, it represents a way reduce the complexity of a set of rules. The necessity of a large number of formulas stating the obvious fact that conditions do not change unless an action changes them is known.

[11]A deeper explanation of this approach can be found in [12].

2.3.5 Rules and Exceptions

Instead using constraints, we could define some exceptions. In the bird's example using the word *abnormal* we meant that an *abnormal* bird is an exception.

Implementing this semantic we notice that the behavior is quite similar to *abductive reasoning*. Indeed, we define some general rules, but if they are contradicted by an exception (it's like to say if a rule does not respect one of the integrity constraint), the rule is withdrawn.

Using Rules & Exception, we usually distinguish between clauses with positive head (*rules*) and clauses with negative head (*exceptions*).

Let's take a look on a new version of the quaker-republican example:

Rules

```
pacifist(X) ← quaker(X)
hawk(X) ← republican(X)
quaker(nixon)
republican(nixon)
```

Exception

```
¬ pacifist(X) ← hawk(X)
¬ hawk(X) ← pacifist(X)
```

As we can see, Exceptions are more specialized than integrity constraints since:

1. they propose a conclusion;
2. we can reason on the specificity of the conditions of a rule and an exception with different conclusions;
3. it's easier transform this model into normal logic programs using NaF[12] [13].

[12]Note that in this model exceptions have the same form of rules, it means that writing them as clauses in Logic Programming Language such as Prolog, we don't need to change syntax or implement different predicates able to evaluate rules and *constraint / exceptions.*

Applying this semantic we can conclude both *pacifist(nixon)* and *hawk(nixon)* as we have two different models (like abductive theories) and they are useful to fix conflicts in order to seek which rule generate contradictions (i.e. In the case "`¬ pacifist(X) ← hawk(X)`" we should withdraw the first clause).

Now we propose an attempt to transform Rules&Exceptions into a Program:

```
pacifist(X) ← quaker(X), not ab1(X)
hawk(X) ← republican(X), not ab2(X)
quaker(nixon)
republican(nixon)
ab1(X) ← hawk(X)
ab2(X) ← pacifist(X)
```

This program has the same models of *nested negation,* but its representation is quite similar to logic programming. In the appendix we can see a brief description of the negation in prolog needed to understand the behavior of a possible implementation of logic programs. Logically, the difference between *negation* and *negation as failure*, can represent an advantage to be exploited in case of lack of informations, but can also cause ambiguous conclusions as we consider *false* something that is *not proved to be true*, rather than *proved to be false*.

2.4 Conclusion

In this chapter we introduced some different logic approaches able to deal with non-monotonic reasoning. We also considered a potential implementation in a logic programming language such as prolog. We evaluated which are the advantages in using a particular representation rather than another one, specially to better define a link between formal representation and language representation, without losing expressivity.

Chapter 3

Argumentation Framework

In this chapter we discuss the fundamentals of argumentation and its possible applications in AI. Analysing argumentation is needed since our purpose is the implementation of a game theory for defeasible reasoning. As described in chapter one, our defeasible (meta)interpreter is just the under-layer of our software architecture. We need to realise a new component, implementing an abstract argumentation framework, able to answer to our specific questions, evaluating results coming from under-layer defeasible Meta Interpreter. Thus, we need to introduce main concepts about argumentation framework (AF), game-theory and in parallel sketch an implementation. In the next sections we discuss some different formal definitions proposed over the years. Special emphasis will be placed on Dung abstract framework, Pollock recursive semantic for defeasible logic and on Rahwan&Larson game-theory.

3.1 A brief introduction to Argumentation

Looking for a definition of *argumentation* on a dictionary, we got two different explanations, the first one is *"the process of arriving at reasons and conclusions; arguing or reasoning",* the second one is *"discussion in which there is disagreement; debate*". The former represents a relation with previous chapter, instead the latter give rise to discuss about the concept of *debate.* Since middle age, this topic has been popular among philosophers, and just then was defined the first formal *debate* between two people arguing opposite conclusions. Abelard, a *Scholasticism* philosopher, in his book *Sic et non [14],* described a *Method*

of issues concerning a debate between two different *ideas*, and a dialogic structure where by adding some arguments it is possible to decide which of the starting ideas is the right conclusion. So we can argue that argumentation is more than "reasoning" since it involves interaction. Deserve a mention also *Aristotle's formal system methodology [15]* and *Plato*'s *Dialogues [16]* that provides the first form of an intelligent interaction among reasoning agent.

Nowadays, argumentation analysis arises the study of non-monotonic reasoning. Indeed, as we saw in the previous chapter, if we try to use classical logic on a more expressive kind of inference, we notice that it is inadequate, and it is due to its intrinsically monotonic behaviour.

Argumentation serves knowledge representation and reasoning as a representational use of logic in Artificial Intelligence. Adding non-monotonic reasoning we mean that a conclusion can be supported by a *defeasible* premise that in turn can be defeated by the attacks by others argument or by new informations. Thus, the obtained conclusion (derived from a chain of defeasible reasons) is not a proof, but an argument.

That is the reason why we sometimes call this kind of reasoning *defeasible Argumentation.* In the following sections[17] we show different kinds of argumentation system proposed over the later years.

3.2 Argument as a kind of logic

The first research work we analyse has been developed by Stepen Toulmin [18]. He introduced a conceptual model of argumentation, in particular, he studied the legal arguments and he distinguished their form in four part:

- claim;
- warrant (non demonstrative reason that allows the claim);
- datum (the evidence needed to use the warrant);
- backing (the ground underlying the reason);

His historical contribution to argumentation deserves the mention since he laid the foundations for argumentation study. The first complete approach is the idea of *defeasibility* into

epistemology and a consequent concept that is *justification* defined by Pollock [19]. He says that reasoning operates in term of reasons. We have two different kinds of reasons: *non-defeasible* and *defeasible;* defeasible reasons are defined as a particular knowledge (*defeaters*). In other words, a *defeater* attacks a conclusion reached by another reason. He defined two kinds of *defeaters*:

rebutting: if the defeater attacks the conclusion of another reason and supports the complementary;

undercutting: if the defeater attacks the relation between a reason and the conclusion (i.e. some premises);

Furthermore, he proposed a new definition of the *warrant* notion. Before, *warranted* conclusion was defined as a conclusion that has an argument for it, now Pollock says that we can consider a conclusion as *warranted* only if after an iterative justificatory process it emerges *undefeated.*

During the history, philosophic logic dealt with knowledge incompleteness and its representation. At times, defeasible logic hasn't been considered as the proper way to survey this topic; at other times has been showed that the analysis done so far was not complete, and it was necessary to investigate some aspects, such as the different kind of conflict.

Deserves a mention D. Nute, who realises the first formalism (LDR1) in which the competition was among rules and not among arguments; in the next years he realised the first implementation in *Quintus Prolog* of an argumentation system in defeasible logic.

Later Dung gave, with his new abstract framework, the major contribution in this field. Dung defined a conjunction ring between argumentation and logical programming. He showed that it is possible to implement argumentation using *Negation as Failure* and introduced a method for generating meta-interpreters for argumentation system [20]. Later, a lot of combined implementation of Dung's and Pollock's system has been defined, in particular we mention the framework realised by Simari in his PhD Thesis, and the following works with Loui.

3.2.1 Concrete applications

3.2.1.1 Acquisition of knowledge

The first application we show is the natural language comprehension. Alvarado, created a framework (OpEd) [21] that after reading some editorial segments concerning specific fields of policy and economics, is able to answer questions about the content. It's important underline the work behind this approach. The argument framework has been built as a network of arguments, whose relations are different kinds of attack or deductive relations etc... He also defined some features related the atomic concept of argument and not the specific proposition viewed as argument or part of it. Thus, it was able to him to develop an abstract framework.

3.2.1.2 Legal reasoning

Another important field of application is the law. In the first time argumentation has been used just for retrieve cases pertinent to a legal argument (BankXX). Later, different approaches have been proposed. We mention J. Huge, that defined an *elegant "reason-based logic*". He stated that application of a rule leads to a reasoning that brings towards the rule's conclusion itself. Other contributions concern using of priorities in order to choose in case of conflict which is the preferred argument, the attempt to model the legal reasoning as a dialogue, and new paradigms such as *legal merit argument* in which the value of a legal outcome is measured in terms of moral, ethic, social welfare, etc...

3.2.1.3 Decision task

One more application field, is the decisions task. This is interesting as have been proposed a lot of different approaches such as defeasible arguments, and also *persuasive* argumentation in order to define tactical rules to reach a goal. It's important underline that here we can define a strategy or a structured defeasible argument that (although not provable) could bring toward the definition of a successful result.

3.3 The study of some argumentation systems

Our aim in this section is to offer a complete survey on the different argumentation framework that have been proposed during the last years. Some of them have been introduced in the previous section such as Dung's and Pollock's systems.

3.3.1 Rescher's *Dialectics*

In [22] Rescher proposed a new formal disputation describing a process. The novelty is that there are three parties : the *proponent,* the *opponent* and the *determiner.* The process starts with the proponent thesis, that is the root of our tree representation and it proceeds using the rules and assertions in order to propose an *answer.* In this system an *answer* is a structured move, namely, a combination of the fundamental moves:

Categorical assertions: represented with *!P* and it means *"P is the case"*.

Caution assertions: represented with $\dagger P$ and it means *"P is compatible with all that have been shown"*.

Provisoed assertions: represented with *P/Q* and it means *"if Q is the case, then **usually** P is the case also"*.

The first move, namely, the thesis proposed at the beginning of the debate, is always a *Categorical assertion.*

The behaviour should be as follows:

1. The answer to a *categorical assertion* as *!P* could be whether $\dagger \sim P$ (a *caution counter-assertion*) or a *provisoed counter-assertions* such as $\sim P/Q\& \dagger Q$.

2. The answer to a *caution assertion* as $\dagger P$ could be whether $! \sim P$ (a *categorical counterassertion*) or a *provisoed counter-assertions* such as $\sim P/Q\& \dagger Q$.

3. The answer to a *provisoed assertion* $\sim P/Q$ must be another *provisoed (counter)assertion* equally or better informed, and it form can be *strong* or *weak:*

   - strong: $\sim P/(Q\&R)\&!(Q\&R)$.
   - weak: $\sim P/(Q\&R)\& \dagger (Q\&R)$.

Provisoed assertion are considered always true, and that is the reason why is not allowed to attack it with the negation or a direct *caution counter-assertion.*

One of the party *concedes* an assertion if it does not answer to the claim of the assertion by the other one.

The game proceeds until one of the parties stops to answer. If the *resources limitation* leads the match to a draw, the third party will decide which is the winner, according to the plausibility of the assertions proposed by the proponent that were not conceded by the other party.

Plausibility is a strong concept and a central mechanism. It is used to explain the idea of *presumption.* In a debate, one party could argue a *presumption* rather than a random proposition. A *presumption* is just a supposition, and it's defeasible. Its goodness depends on the plausibility, namely, how it harmonises with the rest of the knowledge.

Rescher also proposed the concept of "*burden of proof*", that can be distinguished into:

- *probative of an initiating assertion* (the proponent).
- *evidential burden of further reply in the face of contrary considerations.*

This kind of approach explains better the intrinsically progressive process of a debate. Rescher's work has been really useful for a mature definition of the argumentation systems and, maybe, in its framework we can see a forecast of the *Default logic.*

The reason why we mentioned this research work is that Rescher proposed concepts still in use in modern argumentation framework such as relation between fundamental moves[1], limitation of time criteria, and burden of proof, that later will gain importance and will deeply take part on the core of the *reasoning engine*.

3.3.2 Lin & Shoham's argument framework

In [23] Lin & Shoham proposed an *abstract language* (that has been conceived according to the inference rules) for the definition of the following kind of rules:

- A (basic fact) where a is a wff[2];

[1] In the next years moves change their name, but, basically the meaning remained more or less the same.

[2] In mathematical logic, a well-formed formula (often abbreviated wff) is a word (i.e. a finite sequence of symbols from a given alphabet) which is part of a formal language. It is a syntactic object that can be given a semantic meaning. A formal language can be considered to be identical to the set containing all and only its wffs.

- $A_1,\ldots A_n \rightarrow^3$B (monotonic rule) where n>0 and A_i,B is a wff;
- $A_1,\ldots A_n \Rightarrow$B (non-monotonic rule) where n>0 and A_i,B is a wff.

The three kind of inference rules represent respectively *explicit knowledge, deductive knowledge* and *commonsense knowledge.* This abstraction is useful to define the atomics components that build an argument. In the tree-representation, an argument is a root with some labelled arcs. *An argument supports a formula if the formula is the root of the tree that comprises the argument.*

He defined the *argument structure* concept. A set $\mathcal{T}$ of arguments is an *argument structure* if it satisfies the following conditions*:*

1. Let *p* be a basic fact, then $p \in \mathcal{T}$;
2. $\mathcal{T}$ must be closed, if $p \in \mathcal{T}$ and *p'* is a sub-tree of *p,* then, *p' belongs to* $\mathcal{T}$;
3. $\mathcal{T}$ is monotonically closed, if *p* is the conclusion (in a monotonic rule) of some p_i that belong to $\mathcal{T}$, then also p belongs to $\mathcal{T}$;
4. It is consistent, cannot support an element and its complementary.

The latter bring us to the definition of completeness. We say that a set is complete *w.r.t* to a formula ϕ if it supports ϕ or $\neg\phi$.

The power of this framework is that it is abstract and can be used with *default reasoning*, *negation as failure* and *circumscription*. Language definition can be easily implemented in a logic programming and in the implementation chapter we will see a rather similar definition of rules distinguished in *monotonic rule (strict)* and *non monotonic rules (defeasible)*.

3.3.3 Simari & Loui: Pollock and Poole combination

In this section we deal with a mathematical approach introduced in [24] by Simari & Loui. They presented a general system that merges the theory of *warrant* that we saw before in the Pollock framework and the *specificity* concept proposed by Poole. They used the *argument structure* described above, and tried to shed light on the conditions under which a structure is preferred, according only to syntactical considerations.

[3]Note that this notation is not the same to the we presented in the Chapter 2. This notation is needed just to distinguish between monotonic and non-monotonic rules.

They distinguished the knowledge in indefeasible($\mathcal{K}$) and defeasible(Δ), so the whole knowledge has been represented by a pair ($\mathcal{K}$, Δ). The former should be consistent, the latter is a set of defeasible rules expressed in the metalanguage. In order to define a derivation from $\mathcal{K}$ using the ground item in $\cdot$ we need to realise a meta-meta relationship using a formula *h.* Simari & Loui called it *defeasible consequence* and we can see it as material implications for the application of modus ponens.

We generally use a subset of the defeasible knowledge, let $\mathcal{T}$ be a subset of Δ, we say that it is an argument for a sentence *h* if $\mathcal{T}$ derives *h*, $\mathcal{T}$ is consistent (w.r.t $\mathcal{K}$), $\mathcal{T}$ is minimal. In this terms we are able to define a *argument structure* that consists in both $\mathcal{T}$ and *h* (< $\mathcal{T}$,*h*>).

$\mathcal{K}$ usually is divided into two set, the first one called $\mathcal{T}_\mathcal{P}$ represents the contingent knowledge, the other one $\mathcal{T}_\mathcal{G}$ represents the necessary knowledge.

They represent a dispute with a tree-structure. Tree's branches are represented through three different relationships:

Disagree: Two arguments disagree if and only if combining both their sentences with $\mathcal{K}$ we obtain an inconsistency;

Counter-argue: An argument counter-argues another argument if it disagrees with a sub-argument of the second one.

Defeat: An argument A_1 defeats (in the weak and strong sense) another argument A_2if can counter-argues at the literal *h* a subset A_s of the of the second one and A_1 is *strictly more specific* than the sub-argument of $A_2$and it is not related by specificity to A_s.

An argument that is able to defeat another argument is called *defeater.* Furthermore, we talked about specificity, namely, it is implicitly expressed that we are using preference criteria.

Simari and Loui present an inductive definition to explain the process of **justification.** We should remember that Pollock defined different levels where the arguments played a different roles. S&L keep on using this strategy in order to label the arguments as *in* and *out.*

1. In the level 0 all arguments support arguments and interfere argument.

2. We say that an argument is active as a *supporting* argument in a level *n+1* if there are no *interfering* arguments in the level *n* that defeat it.

3. We say that an argument is active as a *interfering* argument in a level *n+1* if there are no active *interfering* arguments in the level *n* that are *strictly more specific* than the first one.

Finally they defined an argument as *justification* if remains accepted as a supporting arguments. This inductive has been extended with the definition of a tree-structure called *dialectical tree (MTDR)* in which is allowed the consideration of cases of *fallacious argumentation.*

The *dialectical tree* has an inductive definition: a single node containing an argument with no defeaters is a dialectical tree; let A is an argument and B,C,D three *defeaters,* in a dialectical tree A be the root and B,C,D are its children; leaves are undefeated nodes. Generally the nodes can be labelled as U-node or D-node, where U and D meaning is, respectively, **U**n-defeated and **D**efeated. If the root is labelled as U-node, then the sentence *h* supported by the root argument can be considered true. We call it *justified belief* and A is said *justified.*

Tree show a game-tree behaviour where two players in turn propose new argument in order to make *justified* or not the root.

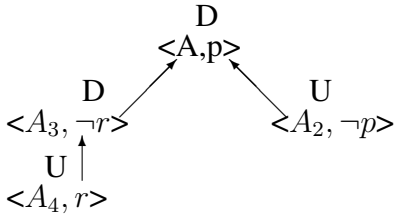


Figure 3.1: A dialectical tree.

In this figure we have the typical representation of a *dialectical-tree.* Despite the meaning of the sentences, we want to underline that the root is defined *justified* if and only if all its children are **U**-node. So in the figure it's obvious that the argument is not *justified.*

A.J. Garcia developed a *Defeasible Logic Programming language (DLP)* using the MTDR as inference procedure.

This work is fundamental for our application since Simari&Luoi defined the tree derivation structure to asses the state of an argument and specificity criteria. In this manner we can

express the first *superiority relation* among arguments. If the tree that proves an argument A contains the the tree that proves an argument B, A can be said "*more specific than b*".

Furthermore, it becomes clear the idea of a game-theory implementation in argumentation debate; this approach is resumed by Vreeswijk whose abstract framework (we wont explain for reason of space) defines also the concept of won for an argument.

3.3.4 Brewka's approach: Rescher and Reiter combined

Their work provides an extension of default logic, in a different form, namely, the provisoed assertions are seen as Reiter's default logic. The proponent's goal is to build a default theory that implies a formula and persuades the opponent to accept it. The opponent, instead, should pose questions or add new evidence against its claim. Also in this case the knowledge is divided into facts(W) and default(D). Brewka proposed two different extensions:'

- PDL (Priority Default Logic): uses the priority to solve conflicts;
- SDL (Specificity Default Logic): use the specificity[4] criterion.

In DL extensions represent set of acceptable beliefs a reasoner might adopt.

His most important contribution, maybe, has been the definition of SLD-extensions, but its definition depends on PDL, so we present a brief summary about a prioritised extension.

First we must define a prioritised default theory as a triple (D,W,<); Given a strict total order "$\gg$" containing "<", we say that $\mathcal{E}$ is a PDL-extension of Δ (default theory) iff $\mathcal{E}$ is the union of some sets built inductively:

$\mathcal{E}_0$ = Theorem(W), and $\mathcal{E}_{i+1}$= $\mathcal{E}_i$ if no default is active in $\mathcal{E}$ or $\mathcal{E}_{i+1}$= Theorem($\mathcal{E}_i \cup$\{b\}) where *b* is the consequence of the "$\gg$"-minimal default that is active in $\mathcal{E}$.

Let's come back to SLD, we say that $\mathcal{E}$ is a SLD-extension of Δ if the specificity ordering "$<_\triangle$" of $\triangle$exists and $\mathcal{E}$ is a prioritised extension of (D,T$\cup$C,$<_\triangle$).

That is an important result as shed light on a relation between specificity and priority, namely, specificity is a particular priority criterion.

Furthermore, he defined a distinction between the default theory of a proponent and the one of the opponent ($\triangle_{pro}$, $\triangle_{opp}$ in order to distinguish the different supported theories), and he re-defined the fundamental moves:

[4]In case of conflict it should prefer the *defaults* more specific to the more general ones.

additions: $add_g(i)$ where *i* is a formula, if g=T (background knowledge) then *i* is added to the current player's background knowledge, otherwise to its contingent knowledge;

concessions: $conceed_g(i)$ moves the formula *i* in the other player's background knowledge;

removals: $remove_g(i)$ removes *i* from the current player's background knowledge.

For a move to be consistent, the current player theory must have at least one consistent extension. To be legal, a move must be consistent and lead us to a new state, never considered before.

We are obliged to underline the distinction of specificity and priority, that conceptually are really different, but, during implementation we will see that can be merged in the same predicates since specificity relation is somewhat a relation of superiority.

3.3.5 Dung's argumentation theory: extending logic programming

The central notions of the Dung's argumentative theory [25, 26] are: *acceptability* and *extended logic program.* The latter is a way to represent a program $\mathcal{P}$ who contains particular clause depending on atomic literals (*objective literals*) and the negation of some other literals (*subjective literals*).

Subjective literals are also called *assumption* and they are the atomic elements on which is based the proof. An argument A supports an objective literal L if there exists a proof of L on assumptions contained in A. When an argument supports both L and ¬L, it is called *self-defeating* and represents inconsistent between its assumptions due to one or more conflicts.

The available ways to attack an argument are:

Reductio Ad Absurdum (RAA-attack) A attacks A' and A ∪ A' us self defeating

Ground Attack (g-attack) A attacks A' and an assumption of A' support $\neg L$, but L is supported by A

According to Dung's theory the semantic of an extended program $\mathcal{P}$ is defined by the semantics of the argumentation framework defined as follows:

$$\mathcal{AF}(\mathcal{P}) = \langle \text{ A, attacks, g-attacks} \rangle$$

where attacks is the union of g-attacks and RAA-attacks.

Now we propose the notion of *acceptability* and *admissibility* according to Dung's theory. Given a conflict-free[5] set of argument S, an argument A is *acceptable* w.r.t. S if for any argument B that attacks A, B is ***g-attacked*** by some argument of S. If all element in S are acceptable, we say that S is admissible.

Dung's framework reveals important role since we will use this abstract formalism as starting point for our work. We will implement the same kinds of attack and the same criteria to determine if a rule is *rebutted/undercutted* or not.

3.3.6 Prakken & Sartor's framework for legal argumentation

In[27] Prakken & Sartor presented a framework for defeasible argumentation that provides two kinds of literals *strong* and *weak. Strong literals* are atomic first-order formula, *weak literals* are the "weak-negation" of the *strong literals*; thus, let L be a *strong literal*, its related weak literal is $\sim$L. Pay attention to the difference between $\sim$ and $\neg$. The former should be read as "*there is no evidence that L is the case*", instead the other one means "L is definitely not the case"[6].Prakker and Sartor introduced this new symbol in order to represent assumptions.

They proposed two argumentation frameworks:

- Framework for *fixed* priorities;
- Framework for *defeasible* priorities.

Framework for *fixed* priorities

Their framework read a theory passed as input called *ordered theory* in which the rules are distinguished into *Defeasible rules (D)* and *Strict rules (S)* that contains only *strong literals.*

An argument is a set of ground instances of rules composed by *strong literals* and there is a ordered sequence of rules and related literals, and a conclusion for an argument A is L, where L denote a consequent of some rules in A.

In this framework is also useful define a concatenation between argument and rules in order to show better the possible cases of conflict between arguments. The possible attacks are:

[5]It means that there are no attacks among its elements.

[6]It is not so far from the Dung's definition of *objective literals* and *subjective literals*, also if Dung used the $\neg$ symbol to denote the subjective literals.

Rebutting Attack A rebuts B if they lead to complementary conclusions and, according the priority criteria by induction, A is preferred.

Undercutting Attack A undercuts B if a conclusion of A is a strong negation of an assumption of B.

The *defeat* notion derives from the attacks discussed above. We say that A_1defeats A_2 if A_1 is empty and A_2 is incoherent or A_1undercuts A_2 or A_1rebut A_2 and A_2 does not undercuts A_1[7].

Notice that it is not mentioned the relation between *defeat* and priority order. So it is a weak definition.

An argument A is acceptable w.r.t. a set of arguments *Args* iff every argument that attacks A is attacked by an argument in *Args*. Thus, given a ordered theory Γ, we can define the set of all *justified* argument $JustArg_\Gamma$in terms of fix-point operators (*characteristic function*) of the ordered theory w.r.t a subset of $JustArg_\Gamma$.

As we saw in the previous section, the *characteristic function* behaves monotonically. Thus, given an argument A and a *ordered theory* Γ:

A is Justified iff A is in the least fixpoint of F_Γ.

A is overruled iff A is attacked by a justified argument.

A is defensible iff A is neither justified nor overruled.

Framework for *defeasible* priorities

In the previous section we defined the argumentation framework as a triple where the third element was the priority relation. Now we want do describe the other approach, namely, the one where priorities are not fixed, but defeasible. We are moving the priorities relation from *meta-level* to *object-level* and that is the reason why in our new definition of the framework we have just two elements that describe it:

$$\mathcal{AF} = \langle\, \mathcal{S}, \mathcal{D} \rangle$$

[7]Notice that it is not possible to counterattack a Rebutting Attack with another Rebutting Attack, as it depends on ***fixed*** priorities and and the counterattack would be inconsistent. Although, a similar approach could be analysed applying *defeasible* or simply *not-fixed* priorities. We will show the *defeasible* priorities approach in the next section.

Furthermore, we add a set of strict rules in every theory in order to define a strict *partial* order and a link between meta-level and object-level. Now the ordering component "<" is determined by *priority arguments*, i.e. (r,r')$\in$ < iff there exists a justified argument for $r \prec r'$, namely, the meta-relation between r and r' depends on the fact that must exist a *justified* argument, that this relation of priority. The engine remains more or less the same, obviously, we have to put in account that also during the conflicts and attacks analysis the priorities are defeasible[8]. Maybe, the most important change regards the monotonicity of the characteristic function; indeed this feature is true just for the conflict-free sets of arguments.

This work can be view as a refinement of Dung's framework in legal arguments. We underline the difference between priority *fixed* and *defeasible.* Later (during implementation explanation) we will show you a deep usage of this theory applied on priority and incompatibility relation that will be considered as normal defeasible rules.

3.4 Extensions and extension-based semantics

So far we talked about arguments evaluation as if it was undeniably unique. Instead, Given some arguments that may attack or be attacked clearly they can not stand all together and in order to know which their status we need to evaluate them. It means that evaluation criteria would assume different numbers of arguments that can stand together[9].

In this section we introduce the semantics. We say that an argumentation semantic *"is the formal definition of a method (either declarative or procedural) ruling the argument evaluation process"*[28]. In the literature, argumentation semantics are distinguished between: ***extension**-based* and ***labelling**-based*.

"In the extension-based approach a semantics definition specifies how to derive from an argumentation framework a set of `extensions`*, where an extension E of an argumentation framework is a subset of the set containing all the arguments, representing a set of arguments which can* "survive together" *or are* "collectively acceptable".

In the labelling-based approach a semantics definition specifies how to derive from an

[8]In this case the authors added a new representation *Args-defeat,* instead *defeat.* This is due to distinguish between a normal defeat and a defeat depending on a *not-fixed* priority criterion.

[9]We remind that an argument is *justified* status if there exists a proof-tree in which the argument is the root (U-labelled) and no one of its children is non D-labelled.

argumentation framework a set of `labellings`*, where a labelling L is the assignment (mapping) to each argument in A of a label taken from a predefined set L, which corresponds to the possible alternative states of an argument in the context of a single labelling".*

3.4.1 Bondarenko, Dung, Kowalski & Toni's abstract AF

In [29] authors discussed sa generalisation of Pool's Theorist [30] framework that allows each theory, that is formulated in a non-monotonic logic to be extended by a *defeasible set of assumptions.*

The aim is to define a framework able to decide if some assumptions can be accepted as an extension for a given theory, or not. In order to define a extension we need to put some constraints, and the extension acceptability depends on the choice of the constraints.

So the question could be: "*Which are the constraints that make us able to understand if an extension is acceptable?*"

Naive semantics The naive semantics are the simplest extensions. An extension is naive, if is maximal and conflict-free. Its greatest feature is that there exists an naive extensions for each assumptions-based-framework that provide at least a conflict-free extension..

Stable semantics As we saw in the previous section, stable semantic are credoulus, in other words they considere more arguments as acceptable than a skeptical semantic. In this context, a set of assumptions is stable if it is closed (in the sense that not contains assumptions that are proved as false),conflict-free, and it attacks all the assumptions it does not contain.

Admissibility and preferential semantics (They are both credoulus) The former extends the admissibility semantic proposed by Dung. It is a relaxation of the stable semantic (same behaviour of preferred semantic saw above). Preferential semantic are really similar, but consideres just the maximal set as admissible; in the sense that the sub-set of the extension are not admissible. We presented those two semantic togheter as they are equivalent if our context is a normal assumption-based framework, and according to the maximal conflict-free notion.

Complete and Skeptical semantics Complete semantics are credoulus, and their constraint is that an extension is complete if it contains all the assumptions that it defends. The last semantic is Skeptical and it accepts as justified only conclusions that can be derived in all acceptable sextensions. It is important as the skeptical version of the stable semantic is the basis of the well-formed semantics of logic programming

3.4.2 Dung semantic analysis

Dung in [31] proposed and compared a set of different extensions. Now we investigate the differences among them. First of all we need to clearly distinguish between the *skeptical* and *credulous* semantics:

Skeptical: is a semantic that considers acceptable an argument only if acceptable for each extension.

Credulous: is a semantic that considers acceptable an argument if it is acceptable at least for one extension.

Roughly speaking a Credulous justifications include Skeptical justifications. Dung defined credulous semantics using the notion of *preferred extension*, namely, it is the **maximal** admissible set of arguments. They are always defined for argumentation framework.

Stable semantic, are stronger semantics, in sense that given a set of argument this is a stable extension iff for each argument A that does not belong to S, there exists a relation (*attack*) between S and A. In other words, S *attacks* every argument that is not contained in S.

It's stronger in the sense that it is more restricted, namely, given a set of argument it is not always possible to define a stable extension:

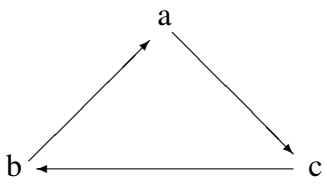


Figure 3.2: Impossible application of a stable semantic

Preferred extension is a relaxation of the constraint imposed in the stable semantic. It means that it is not needed to attack all the argument not contained in the extension, but just the ones that attack an argument that belongs to the extension. Thus a stable semantic is also a preferred semantic, but it's not true the converse.

Dung introduced a new kind of semantics called *ground semantics.* Those are the typical *Skeptical semantics*, and a unique-state, it means that for each argumentation framework is it possible to get just one extension. It is based on the concept of *fixed-point.* The definition of a Characteristic function allows us to define the acceptable argument in a monotonic way, and the least fixed-point of this function corresponds to our *Ground Extension.*

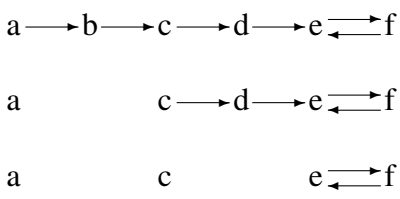


Figure 3.3: Typical ground extension and iterative approach.

In the Figure 3, there is a simple representation of the possible iterative approach to select the argument of the *ground extension.* The first element is "*a*" as it is not attacked by nobody. So we remove all the argument attacked by "*a*" as they will be considered certainly defeated, and so on. The *ground extension* is {*a,c*}.

The last semantic we see is the *complete semantic*; perhaps, it should have been the first, but we would underline the important meaning of this semantic in relation with the other ones assessed above. Indeed, the complete extension is the link between *preferred extension* (Credulous) and *grounded extension* (Skeptical). Let's define a $\mathcal{CO}$, a set of argument S is a complete extension iff acceptable argument w.r.t. S belongs to S.

Dung proved that a preferred extension can be defined as maximal complete extensions, and the grounded extension is a least complete extension.

For our purpose it becomes really interesting since using the same meta-interpreter and the same defeasible logic, we get different conclusions depending on the semantic we are using. Thus, in some contexts with a *credulous semantic* we couldn't be satisfied for the results, but switching to a *skeptical semantic* we can disambiguate the conclusions. On the other hand if a *skeptical semantic* is to *strict* to get a solution, it will be necessary to switch

to a *credulous semantic.*

3.5 Game model

In this section we discuss how to set up a legal argumentation adversarial as a dialectic process using a meta-logic defeasible framework. What we need is a formal model in order to analyze computationally the argumentation's. To assess the dispute between two parties we use the mechanism of the game as a challenge between two players. Thus, we survey which are the conditions to obtains wins or losses. Computationally, logic is represented by Horn clauses. This approach provides a symbolic knowledge representation with a formally well-based semantics.

We model the argumentative process in a meta-logic setting with a background knowledge that represent the object language clauses encoded as terms. The sequential nature of the legal disputes leads us to use a tree-analysis to survey the correctness and to know which are the chances of win and to find possible strategies.

3.5.1 The meta-logic argumentation framework

Let's consider a dispute between two persons. They will follow a specific schema (i.e. expressions of facts, laws, evidence) according to a given logic. Who start the debate, called proponent, proposes his key-claim and he has to defense it from the possible attacks of the opponent.

> The game is won for a party in the state Si, if some admissible act leads to lost for the other party in a subsequent state Si+1...the game is lost for a party when all admissible acts recursively lead to won for the other party.

The proponent wins if:

- the key-claim is proved in the state Si;
- the key-claim is defeasibly defended in the state Si and the opponent has not more admissible moves in the next state.

The proponent loses if:

- the negation of the key-claim is proved;
- in the state Si it's not proved the negation of the key-claim and the proponent has exhausted his moves.

Accordingly, the opponent wins if:

- the negation of the key-claim is proved in the state Si.
- the negation of the key-claim is defeasibly defended in the state Si and the proponent has exhausted his moves.

The opponent loses if:

- in the state Si the Key-claim is proved;
- the key-claim is defeasibly defended in the state Si and the opponent has exhausted his moves.

Keep in mind that it's important to distinguish the quantifier *some* and *all*. I mean that:

> "A loss means that ***all*** the acts lead to won for the proponent."
>
> "A win means that ***some*** act leads to lose for the opponent."

In the following, we present a defeasible logic prover, described introducing the evolving argumentative state S = {S1,S2,..,Sn} where n < ∞ (So it's a finite set):

1. $\frac{Y1,...,Yn \rightarrow X, S \vdash Y1, S \vdash Y1, .., S \vdash Yn}{S \vdash X}$, it means that X is proved if X is a conclusion of some premises that can be proved by S;
2. $\frac{S \backsim X}{S \vdash X}$ if X is proved, then it's also defeasibly defended;
3. $\frac{S \neg \vdash \neg X, Y1,...,Yn \Rightarrow X, S \backsim Y1,..,S \backsim Yn}{S \backsim X}$ that means: X is defeasibly defended if it's not proved X negation and if X is the conclusions that S can be defeasibly defended.

3.5.2 A simple legal dispute: Assessing the pleas

First of all, we have to define a legal code formalization. Thus, we start with a general analysis. In other words, we need something that is considered by all true.

In the last years a lot of people have been disappointed by some messages concerning licenses check on their personal computer from Windows O.S. Indeed, using that operating system they noticed that there are some applications able to control if for the installed software the customer has purchased a regular license. So, there are two cases *(i)* customer bought computer with pre-installed OS from an authorized seller *(ii)* customer brought computer from a non-authorized shop. In the former case at the first bootstrap, operating system requires that user accepts the license, in the latter case it doesn't happen since it has been installed a pre-configured *illegal* copy of windows and so also if the user didn't accept the license is guilty because he is running a fake copy of the software. Nevertheless, they sue windows for privacy violation.

Legal code (defined by some rules):

Receive_messages $\rightsquigarrow$ privacy_violation

Receiving_messages $\wedge$ accepted_license $\rightsquigarrow$ not(privacy_violation)

User:

Receive_messages $\Rightarrow$ privacy_violation

$\Rightarrow$ Receiving Messages

$\rightarrow$ OS Pre-Installed $\supseteq \neg \vdash$ Receiving Messages $\wedge$ Accepted License $\rightsquigarrow$ not(privacy_violation)

Microsoft:

Receiving Messages $\wedge$ Accepted License $\Rightarrow \neg$privacy_violation

$\Rightarrow \neg$Receiving Messages

$\rightarrow$Accepted License

$\rightarrow$Fake Vendor

A complete meta-logic formalization can be consulted here [32], or a simple attempt for this special case in appendix on page 128.

3.5.3 Running the argumentation game

A running game example could be the following one:

The proponent (the windows user) say:

Receiving Messages $\Rightarrow$ Privacy violation

⇒ Receiving messages
Now Microsoft has got two options:

Speech act opt. A	Speech act opt. B
⇒ ¬Receiving messages (becomes a draw)	→accepted license "You accepted the license when you installed our product."Now proponent could say →OS Pre-Installed ("when I bought my computer the operating system was already installed"). →Finally Microsoft say Fake Vendor

Table 3.1: Game-tree sample

When Microsoft says "Fake Vendor" wins the challenge because the key-claim "Privacy violation" has not been proved, and after its last answer, the proposer hasn't got more acts.

3.6 Conclusion

In this chapter we discussed about argumentation framework and its different implementation proposed over the years, we accepted Dung's abstract framework as starting point for our architecture of arguments. Furthermore, we resumed some aspects introduced later in the years as *defeasible priority, specificity as priority relation* and we defined a game-model in order to realize a simulation of a debate.

Chapter 4

Meta Languages and Meta Interpreters

In this chapter we discuss meta-languages and their representations, later we provide a description of their basic implementation and an analysis of their pro and con. Prefix Meta (from Greek: "beyond"), is used to indicate a concept which is an abstraction from another concept, used to complete or add to the latter, for example, meta-data are data about data. In this context meta represents an abstract level for our language (generally human language) whose has the job of evaluating the object language.

Paradoxes and anatomies are typical cases in which we try to evaluate a language without use an abstract level, thus we reach unacceptable conclusions using acceptable premises and reasoning. The most popular is the "liar paradox" by Epimenide[1] *but also more paradoxes have been studied during the history. Deserve a mention Russel's antinomy*[2]*, whose definition can be summarised so :* "Let A be the set containing all the sets that don't belong to them self. Does A belong to itself?". *This paradox would demand a more specific explanation but for reason of space we omit it and we refer to this document [33].*

[1] Actually, its paradox was quite different. He said that: "*The Cretans are always liars*" and since *Epimenide* was from Crete, it sounds like a paradox. Thus, that sentence is not a paradox. Diogene Laerzio ascribed paradox invention to Eubulides of Miletus that said "*I'm laying*".

[2] The following formula is a simple attempt to show you mathematically this paradox : R={x: $x \notin x$}. Then $R \in R$ iff $R \notin R$.

4.1 Representations

In this section we introduce two ways to represent a language: *ground* and *non-ground.*

> *"if $\mathcal{L}$ is a typed-free language, and $\mathcal{L}'$ is a typed-language that inspects $\mathcal{L}$, then:*
>
> *in a ground interpreter a variable **x** in $\mathcal{L}$, correspond to a constant **a'** in $\mathcal{L}'$."[34]*
>
> *"if $\mathcal{L}$ is a typed-free language, and $\mathcal{L}'$ is a typed-language that inspects $\mathcal{L}$, then:*
>
> *in a non-ground interpreter a variable **x** in $\mathcal{L}$, correspond to a variable **x'** in $\mathcal{L}'$."[34]*

Roughly speaking, the typed language hasn't a correspondence variable-to-variable with the typed-free language, namely, we set a relation from a variable to a constant, and we associate this constant to an index in the typed-language.

Our aim in this chapter is to discuss how to analyze *a language* using *a meta-language*, however, sometimes it could be useless (*easier*) using the *same* language. In other words, we would like to realize a software using a language X and evaluate it with X language itself. This approach doesn't sound so strange since, usually, we describe a language using itself:

"Stockholm is a nine-letter word"

If this is a sentence of a meta-language describing the form of words in an object language, then "Stockholm" denotes itself, and the sentence is true, However, if "Stockholm" denotes the *capital*, the statement is false.[35]

Thus, our issue is finding a way to describe variables, constant and the other terms from a typed-free language to our meta-language. In the followings sections we analyze two different terms representations: *Ground Representation* and *Non-Ground Representation*.

4.1.1 A definition for Meta-Interpreter

An interpreter, informally, is a program that evaluates programs. Interpretation is pervasive both a theoretical point of view than from a practical point of view. A lot of programs are interpreters for specific language domains. A program that read a configuration file for the

set up, has to interpret the language used to write that file. An interpreter for a language that is written with the same or with a similar language, is called Meta-Interpreter. If an interpreter is able to interpret itself, it is called Meta-Circular. Applying this meta-interpretation action in computer science we get a new definition since an interpreter corresponds to a program:

> "*Meta-programming is a programming technique that enables manipulation with program structures. Because Prolog uses the same data structures to represent programs as well as data, Prolog is suitable for writing meta-programs."[36]*

4.1.2 Ground Representation

A possible ground-representation may look as follows:

- each constant of the object-language is represented by a costant of the meta-language;
- each **variable** of the object-language is represented by a **costant** of the meta-language;
- each n-ary functor of the object-language is represented by a unique n-ary functor of the meta-language;
- each n-ary **predicate** symbol of the object-language is represented by a unique **functor** of the meta-language (with the corresponding arity).

Thus we can define a function ϕ between the two domains and its inverse ϕ^{-1} to determine the interpretation of a symbol. Let c is a constant of the meta-language, its meaning $c_{\Im}$ is the constant or *variable* $\phi^{-1}(c)$ of the object-language. Instead, about the n-ary functions we propose some rules:

The meaning $f_{\Im}$ of a n-ary function, is a function that maps:

(i) the terms $t_1, \ldots ,t_n$ to the term $\phi^{-1}(f)$ $(t_1, \ldots ,t_n)$ if $\phi^{-1}(f)$ is a functor of the object-level;

(ii) the terms $t_1, \ldots ,t_n$ to the atom $\phi^{-1}(f)$ $(t_1, \ldots ,t_n)$ if $\phi^{-1}(f)$ is a predicate of the object-level;

(iii) the formulas $f, \ldots, f_n$ to the formula $\phi^{-1}(f)$ $(f, \ldots, f_n)$ if $\phi^{-1}(f)$ is a ***connective***[3] of the object-level;

The next example can help us to understand what we stated above.

Let $\mathcal{L}'$ be our object-language composed of constants *a,b,* the predicates *p/1, q/2*, the connectives $\leftarrow, \wedge$ and an infinite but enumerable set of variable *X*.

Our meta-language $\mathcal{L}$ consists of the constants *a,b,x* the functors[4] *p/1,q/2, and/2* and *if/2*.

If we analyze this sentence in the meta-language:

if(p(x)),and(q(x,a),p(b))

the assigned meaning in the object-language is:

$p(X) \leftarrow^{5} q(X,a) \wedge p(b)$

What we need is a description of the relation between the terms and formulas of the object-language in the meta-language.

```
step(Goal, NewGoal)←

    select(Goal,Left,Selected,Right),
    clause(Selected),
    rename(Selected,Goal,Head,Body),
    unify(Head,Selected,Mgu),
    combine(Left,Body,Right,TmpGoal),
    apply(Mgu,TmpGoal,NewGoal).
```

This pseudo-code, represents a meta interpreter for our object-language. If we use prolog, we would add some predicates to replace the ones built-in in prolog-engine.

(1) takes the goal and selects a sub-goal and its neighbors, we mean the one before it and the next one.

[3] We mean operator such as *and*, *or*, *if*.

[4] Note that the predicates in the object language will be inspected by meta-language using functors representation.

[5] Note that we used the last rule for the *if* formula, as it is a connective in the object-language.

(2) we check if it is a clause that we know (see later).

(3) we rename the clause in order to obtain a variant of *Head* and *Body* of *Selected*, without variables in common with the *Goal* .

(4) gets the mgu of *Head* and *Selected*.

(5) creates a new temporary goal composed by a conjunction of *Left*, *Body* and *Right*.

(6) with *apply* we apply a substitution of the temporary goal *TmpGoal* using the *Mgu* and we obtain a *NewGoal.*

So, we have to save[6] all clauses of the object-language as facts in the meta-language:

```
z:-demo(
[(pred(foo,[var(s(0)),var(s(s(0)))])<=                                    (1)

    [pred(bar,[var(s(0)),const(b)]),
    pred(doe,[const(c),var(s(s(0)))])]),

(pred(bar,[const(a),const(b)])<=[]),                                      (2)
(pred(doe,[const(c),const(d)])<=[])],                                     (3)
[pred(foo,[const(a),const(d)])])                                          Goal
```

So all the goals of the object-language were passed during the goal invocation *foo*(1), *bar*(2), *doe*(3). The second arguments is the real goal, and our meta-interpreter has to control if the invoked predicate (in this case *foo*) is contained into the list of the available goal, and if so, check if it unifies with the goal.

An alternative way could be defining into the program some facts, say, the object-language predicates:

```
clause(⌜(pred(foo,[var(s(0)),var(s(s(0)))])<=

    [pred(bar,[var(s(0)),const(b)]),
    pred(doe,[const(c),var(s(s(0)))])])⌝).

clause(⌜pred(bar,[const(a),const(b)])<=[]⌝).
clause(⌜pred(doe,[const(c),const(d)])<=[]⌝).
```

[6]Or pass them in a list as a sort of dictionary where to store all our object-language information.

A goal such as *clause(X)* shows all the available clauses in the program:

```
?- clause(X)[7].
X = pred(foo, [var(s(0)), var(s(s(0)))])<=...
X = pred(bar, [const(a), const(b)])<=[] ;
X = pred(doe, [const(c), const(d)])<=[].
```

That's a *static* definition, we are not able to execute the predicate inside the clause, but we know that it is correct for the object language.

If we want manipulate a goal, i.e. we need to separate the body by the head or select a sub-goal in the body, we need to define a new clause using the operator **<=** defined before:

```
deeperInvestigation(Head <= Body,...)  <= (do something)
```

It's important investigate the sub-goal in a body of the object-language, for example to define a derivation tree. Thus, using this representation for the clause of the object-language and defining a new predicate for checking the derivation sequence, we are able to know which steps are needed to carry out a derivation-SLD tree for a specific goal:

```
derivation(G, G).
derivation(G0, G2) :- step(G0, G1), derivation(G1, G2).
```

In our case:

```
derivation(⌜pred(foo,[const(a),var(s(s(0)))]⌝,true).  %is there a
derivation from this goal to the empty goal?
```

It matches with the second clause of derivation/2:

```
derivation(...)  :- step(⌜[pred(bar,[const(a),const(b)]),
pred(doe,[const(c),var(s(s(0)))])])⌝, G1), derivation(G1, G2).
```

And so on...till arrive to the empty goal.

[7] It is needed to define the new operator **<=** in the prolog program, putting at the head of the file the following line: `:- op(1,xfx,(<=)).`

4.1.3 Non-Ground Representation

Ground interpreters are more readable since they don't use only buit-in predicate to define unification, substitution and there are no ambiguity between variables that belongs to *object-level* and to *meta-level* but they need more memory to save all the terms of the object language; furthermore, developing a program is a demanding task because we need to rewrite a lot of code.

The non-ground representation is more pragmatical, and *less logical* since they behave in a strange way when we deal with renaming and binding of a variable[8], but allows us to develop interpreters in an easier way. The core idea is to map object-language variables to meta-language variables, instead constants. It straightforward, but there are some semantic restriction.

A typical example used to compare the two meta-interpreters is the one about relationship between relatives:

Example 1

Ground Representation:

```
clause(⌜grandparent(X, Z) ← parent(X, Y ), parent(Y, Z)⌝).
clause(⌜parent(X, Y ) ← father(X, Y )⌝).
clause(⌜father(adam, bill))⌝).
clause(⌜father(bill, cathy)⌝).
```

Non-Ground Representation:

```
clause(grandparent(X, Z) if parent(X, Y ) and parent(Y, Z)).
clause(parent(X, Y ) if father(X, Y )).
clause(father(adam, bill)) if true).
clause(father(bill, cathy) if true).
```

The idea is to represent the whole object-language in the meta-level in order to manipulate and execute object-predicate without "translate" them.

Thus, we don't need rather complicated definitions for the built-in logical predicate about unification, renaming,...etc and the meta-interpreter is quite easier:

```
solve(true).                                                    (1)
```

```
solve(X and Y) ←solve(X), solve(Y).                                    (2)
solve(X)←clause(X if Y), solve(Y).
```

This meta-interpreter is known as *Vanilla Meta-Interpreter* (a deeper analysis with code example is provided A.4)

Roughly speaking, the "translation" is performed in the meta-level, it's like to say that

> *"the meta-interpreter knows the* `meaning` *of the* `connectives` *and* `variables` *of the object language and does not need to build a dictionary to convert the representation in the meta-level with the* `meaning` *in the object-level"*

Coming back to the previous example, a possible goal is ←`solve(parent(adam,bill))`.

This goal matches (3) and its body after the substitution is:

←`clause(parent(adam,bill) if` Y_0`), solve(`Y_0`).`

and this new goal matches with second clause of the Example 1. So, Y_0 `= father(adam,bill).`

So the unification between the two goal is in the meta-level. Furthermore, if we invoke a goal with a variable such as `parent(adam,bill)` prolog[9] returns the result `X = bill,` as renaming predicate used in the meta-level does not need a explicit different definition as in the ground-interpreter, *now X is a variable also in the meta-level*.

4.1.4 *Why do we use Ground MI if they are so complex?*

The reason why sometimes we have to use a Ground-MI is that Non-Ground MIs although are really easy to develop, they haven't a clear declarative appearance. For example, if we consider *renaming*, in non-ground MI we can implement it using the underlying mechanism for renaming, namely, using the *clause* predicate to get the *body*. The disadvantage of this method is that the object-program is fixed, making it impossible to do "*dynamic meta-programming*".

Let *p(X,a)* is a clause that unify with *p(Y,Y)*. after the invocation of the assignment, say, *p(X,a) = p(Y,Y)* both atoms will take the form *p(a,a),* namely, the old *version* of *p* predicate is no longer accessible. Unfortunately, this task becomes a problem while implementing a non-ground meta-interpreter. It would use the same predicate *"in a different ways on*

[8] In the next section those kind of issues will be better explained.

[9] Prolog is our point of reference.

the same branch". It means that it cannot evaluate decoratively a code with this particular issue[10]. One more problem is that we are not able to test in a *declarative way,* whether two atoms are variants (or an instance) of each other. Furthermore they express ***incorrectly*** the quantifier. Let a program contains the following code:

```
p(X)←q(X). % ∀X [p(X)←q(X)]
```

in Vanilla MI it means:

```
demo(p(X)←q(X)). % ∀X demo(p(X)←q(X)).
```

that is different as it means that for every term *t* the clause belong to program.

Thus, solving those problems it's a task that is entirely up to the programmers, however, since those behavior are also due to the fact that Prolog is not completely declarative, and could be necessary to switch to a different kinds (*strongly typed*) of programming language such as *Göedel*[11] .

4.2 Conclusion

In this chapter we got acquainted with the most popular representations of a language : *ground representation* and *non-ground representation.* Later we implemented and tested those two different approaches on the same programming language (*Prolog*), finally we discussed about different applications depending on their strengths and weakness.

In the following chapter we will show you our implementation of a non-ground meta-interpreter (*Vanilla-like*) since we evaluated that our application would not be affected by non-ground limitations.

[10] Actually, a partial implementation could be performed using prolog built-in predicate *copy/2*.

[11] We tried to test the behavior of our software on Göedel, but we had problem to use it. It is not more updated since 1996 and it is capable only with Unix system. Nevertheless, we had problem running the software on Ubuntu Linux 10.04. Could be interesting in future works analyzing a new implementations of the same meta interpreter on a *strongly typed* language such as *Göedel.*

Chapter 5

Implementation

In this chapter we present our implementation. In the first part we describe the underlayered defeasible meta-interpreter, later the game-model we use to perform the debate between two parties. Some issues caused by resources limitation and some suggestions for future work are described in the last sections. We decided to use Sicstus Prolog for meta-interpreter implementation and Java Rich Client Platform to provide a simple graphical interface. In the following sections we use two friendly names to distinguish between meta-interpreter, that we call Maggie and the graphical interface, that we call Ubongo[1].

5.1 Getting started

Our aim is to build a non-ground meta-interpreter using defeasible logic. This task involves investigating prolog features and conclude if it is able to perform this process. We know that defeasible reasoning is just non-monotonic reasoning and prolog is suitable for this task, indeed adding new informations (new clauses) in a prolog program the result for a goal could change. Unfortunately, we have problems when we try to represent assumptions and defeasible conclusions since Prolog has not been developed according to this kind of behavior; it usually answers saying *definitely yes* or *definitely no*[2]. Prolog rules are not

[1]Maggie and Bongo are my girlfriend's dogs, nevertheless, we called the GUI Ubongo as funny thanks to GNU/Linux (not only Ubuntu) and Free Software community that provide a lot of excellent tools for developers. All this thesis implementation, except for Sicstus Prolog, has been implemented only using free software.

[2]That is not properly true since applying *closed world assumptions* or *circumscription,* in other words, using *Negation as Failure* prolog answers also when it does not contain enough information to be completely

defeasible and so what we need is a new syntax in order to expand prolog expressivity and make it able to allow new kinds of knowledge representation.

Thus, we defined a new kind of rules (*defeasible rules*) those rules are usually true, but when we evidence that we are dealing with an exception, they can be defeated. One more problem is the negation. In the previous sections we talked about *negation as failure* and *negation,* and we noticed that prolog doesn't provide a negation predicate. A wrong point of view could bring us to argue that the problem is not in prolog, we mean that the rules (not defeasible) still work, but there is something, maybe not explicitly written in the code, that blocks them. But it's not so. A typical example [37] to explain this wrong approach is the following one:

Let say Wolf and Fox are all running for the same elective office. Someone might say, "I presume Wolf will be elected, but if he isn't, then Fox will be." How could we represent this assertion in Prolog?

```
will_be_elected(wolf).  %fact
will_be_elected(fox):- \+ will_be_elected(wolf).  %clause
```

But it is different, and also replacing the fact with a new clause:

```
will_be_elected(wolf):- \+ will_be_elected(fox).
```

Taking into account the latter clause, we notice that there are not enough informations, and a goal as *`will_be_elected(X).`* could returns an error or generates an infinite loop depending on which prolog engine we are using.

Let's add some detail to our example. Suppose our political prognosticator goes on to say, "Of course, Wolf won't win the election if Bull withdraws and throws his support to Fox. "How shall we represent this in Prolog?

```
\+ will_be_elected(wolf):- withdraw(bull), support(bull,fox).
```

But this isn't a well formed prolog clause. So we need to invent a new kind of defeasible rule and also a negation operator. Indeed, NaF states that *if something is not proved then is the case,* but this is not enough, we need a new operator able to answer true when *something is not the case*.

In literature, usually, has been used a symbol representation to define *defeasible rule* ":=/2" operator, i.e. replacing this operator to the bird's example we have:

```
flies(X) :- bird(X).[3] % X flies as it is a bird
```

sure.

[3]Normal prolog inference will be used only for ***strict*** rules.

```
flies(X) := bird(X).%X should flies, since birds usually fly.
```

The negation we are introducing is denoted by neg/1. One more interesting features about it, is that we can put it at the head of a clause (that is not allowed in a normal prolog clause using NaF).

For the assumptions, we keep on using ":=" symbol. Indeed, an assumption could be seen as a defeasible fact; a fact could be seen as a rule whose body is simply *true*. Likewise a defeasible fact will be represented as follows:

```
will_be_elected(wolf):= true.
```

What happens if our conclusion is defeasible? Or better, if a conclusion depends on weak defeasible facts (*assumptions*), what should our prolog engine do? Actually, it should declare a *draw* as is not possible to hold that the conclusion is true, nor that it doesn't. This conclusion sheds light on a problem: the attacks we studied in the defeasible logic, can be represented in prolog programming in the same way; we mean a *rebutting attack* when a defeasible conclusion is attacked by another defeasible rule that supports the opposite conclusion, and an *undercutting-attack* that attacks not the conclusion, but an assumption that leads a rule to a *defeasible conclusion*[4]. The representation of a defeater is usually (in literature) performed with the following symbol ":^". Thus, if we wanted to express that Wolf might not win the election if voter turn out is poor, we can use the just described representation as follow:

```
neg will_be_elected(wolf):^ poor_voter_turn_out.
```
[5]

So far, we show two different rules (*strict* and *defeasible*). But we didn't mentioned in which way a conclusion can be derived nor what to do in case of conflicts.

First, we define two different kinds of derivation for a conclusion:

Strictly(*derivable*) if the conclusion derives only from strict facts and rules[6].

Defeasibly(*derivable*) if the conclusion derives also from non-strict facts or rules.

[4]Usually the defeasible rules that *under-attack* another defeasible rule are called ***defeaters.***

[5]Notice that this is not a rule with a conclusion, we are not saying that if the voter turn-out is poor, then wolf will not be elected, but we are saying that if the voter turn-out is poor then Wolf might not win the election, that is quite different as the latter is not providing a conclusion.

[6]We say that a conclusion strictly derivable is also *at least* defeasibly derivable.

In the previous chapters we dealt examples in which were not obvious conflicts between strict rules; but normally we have them. Given two strict rules:

```
mammal(X) :- dolphin(X).
neg mammal(X) :- fish(X).
```

and more rules that evidence that an animal is a *dolphin* but it is also a *fish*, we conclude both is and isn't a mammal, it means that we can't solve this conflict as both are defeasibly derivable. The solution is to refrain from conclude one of them[7]. One more typical example in the literature is:

```
native_speaker(X,german_dialect) :- native_speaker(X, pa_dutch).
born(X, Pennsylvania) := native_speaker(X,pa_dutch).
born(X,usa) :- born(X, Pennsylvania).
neg born(X,usa) := native_speaker(X,german_dialect).
native_speaker(hans, pa_dutch).
```

In Pennsylvania there exists a dutch dialect called "*pa dutch*", who speaks this dialect ***usually*** was born in Pennsylvania, namely, was born in Usa. But, as people who speak dutch dialect ***usually*** were not born in Usa we reach a conflict state. As we can see both the conclusion depend on a *defeasibly derivation.* Which one is to be accepted?

Also if the condition of the strict rule is defeasible derivable while the second one is strictly derivable, the strict one is superior. In this case we assume that Hans was born in the Usa (according to the following rule):

> *a strict rule can only be defeated if its condition is only defeasibly derivable, and then only by a fact or by another strict rule that rebuts it.*

In other words the strict rule is superior, but adding more information able to defeat its defeasible condition, leads us to withdraw the conclusion and to accept the opposite one.

Coming back to the example of mammals, we should explain better how to avoid this kind of issues. Literature is full of similar example [8] in which we can conclude contrary

[7] After all, we analyzed this problem in terms of ambiguity we defined two approaches: *blocking ambiguity* and *propagation ambiguity*.

[8] In particular the example of incompatibility between capitalist and marxist. Ping is from china ans *usually* people from China are marxist, but she left China to go to Usa and now she owns a restaurant, but a restaurant owner *usually* is a capitalist. How can we disambiguate those clauses?

conclusions. It is due by the fact that we should add a new prolog predicates in order to express the disambiguation.

```
incompatible(capitalist(X),marxist(X)).                    (5)
```

Thus, we can conclude that two clause *conflict* or *are competitors* if their heads are contraries, but also defining a predicate *incompatible/2* able to tell to d-Prolog[9] that two clause could be incompatible also if they aren't complements.

In our case, we should implement a contrary relation between two rules, and we can do it using this code:

```
contrary(Clause1,Clause2):- incompatible(Clause1,Clause2).
contrary(Clause1,Clause2):- comp(Clause1,Clause2).
comp(neg Atom, Atom[10]).
comp(Atom, neg Atom).
```

In other words, both conclusions are defeasibly derivable, but logically, for a normal man, those conclusions are in conflict. It means that we need two add some new informations, as prolog without them cannot perceive by intuition this logical conflict.

Furthermore, in this work we added one more defeasible behavior. Since also a compatibility could be seen as defeasible, we decided to propagate a possible definition of defeasible to make it defeasible in some cases:

```
incompatible(marxist(X), capitalist(X)):= point_of_view(strict).
neg incompatible(marxist(X), capitalist(X)):^
    point_of_view(strict),we_are_in_2010.
```

This example sheds light on a deep characterization of our defeasibility. As we can see not only we provide an *incompatible predicate,* but we also use it as it was a normal rule, so it is defeasible and the system also correctly evaluate its negation. So we can assert a defeasible incompatibility, and also using a defeater incompatibility.

[9]Usually with d-Prolog we intend the approach used in prolog programming to implement programs in a defeasible logic.

[10]This definition of Atom is not we mean for atom in prolog, is just a logical atom, namely, a part of a clause, in our case is the head.

A complementary approach[11] is to define some priorities between rules, in particular between two rules that we know could conflict. In our case we use a *sup/2* predicate that tells to d-Prolog that in case of conflict the former is to be acceptable:

```
sup((capitalist(X) :=

    owns(X,Restaurant)), (marxist(X):= born(X,prc[12]))).
```

In this manner the program is able to evaluate *a defeasibly derivable* conclusion: *capitalist(Ping).* Namely, we believe intuitively that if someone escapes from his birth place to open a business activity in order to earn more money, he is capitalist.

We saw in the previous summaries, that Reiter proposed the idea of *Specificity* as criterion to choose between two conflicting rules.

Specificity bring us to choose the more specific rule, and leave aside the more general ones.

In the typical *Bird's* example:

> "*The converse (of the rule* `bird(X) :- penguin(X).`*), of course, is not true: many birds are not penguins. So any rule for penguins is more specific than any rule for birds. Penguin rules take into account more information than do bird rules. When one rule is more specific than another, the first rule is superior to the second. How can we tell that penguin rules are more specific than bird rules? By noticing that the strict rule* `bird(X) :- penguin(X).` *is in our database"[37].*

This problem sometimes can became really difficult to be solved:

```
adult(X) := college_students(X).

neg employed(X) := college_students(X).

neg self_supporting(X) := college_student(X).

employed(X) := adult(X).

self_supporting(X) := employed(X).
```

[11] Those approaches are different, but they can be used together because they cope to different kind of issues.

[12] People's Republic of China.

```
college_student(jane).
employed(jane).
```

In this example we notice that there is a conflict as Jane being a college student she should not be self supporting, but because she is employed she should be self supporting. Which rule is more specific? In this case is not so easy to determine which rule is more specific because both conclusion could be defeasible derivable and there are no obvious clues to leave aside one of them.

So we conclude that we need some investigating rules and some distinction between the set of rules that we should use or not. In particular we call *knowledge bases* those different sets of information that we are using as a part of the whole rules at a given time. And *root Knowledge Base* the entire d-Prolog database. Furthermore, as during the derivation process, *proponent* and *proposer* will add some new informations, we need to keep track of the KB in use at each stage of reasoning.

Our system provides all those approaches: *specificity*, *superiority relation* and *incompatibility*.

5.1.1 Reaching a conclusion

Rather than reasoning about how we explicitly derive a conclusions from some rules, is the case to discuss in this chapter what we discussed above about semantic issues and present a solution for ambiguity dealing.

We defined two families of semantics: *skeptical* and *credulous* semantics. Let suppose that we have some extensions containing some different and complementary conclusion. Using a skeptical semantic we define a set containing the intersection of all the conclusions belonging to extensions. Instead, using a credulous semantic we define a set containing the conjunction of all the conclusions. Coming back to our meta-interpreter this different semantic approach could bring us to different scenarios. Let suppose we have a *goal* and some *rules* that could conclude both *goal* and *neg goal:*

skeptical both conclusions will be rebutted. Maybe, it is better to consider that the intersection of the extensions containing *goal* and the other set containing *neg goal* is empty. According to Nute's first implementation and as a respect for the author, we keep on calling the state as "*I am not able to draw a conclusion*".

credulous both conclusions will be considered true. It means that implementing a debate between two parties and assuming in two different cases that the keyclaim is *goal* and later is *neg goal*, we let both players winning. This example sheds light on some problems about game model that we will see later: "*Does the opponent win if the keyclaim isn't defeasibly derivable or if the negation of the keyclaim is defeasibly derivable or both?*"

Since this semantic distinction, theoretically, deals just the conclusion of a derivation tree, it would be better implement a more general definition of ambiguities more suitable for programming logic. So we decided to implement two predicates that provide the assertion of *ambiguity_propagation* or *ambiguity_blocking.*

Sometimes, ambiguities emerges during the keyclaim evaluation. It means that in the derivation tree we can have one or more nodes that are both *defeasibly true* and *defeasibly false.* It sounds really similar to *skeptical/credulous* semantics and it is. Also if the results are rather similar, it is important to distinguish which approach are we using. Indeed, semantic check could be verified only at the end of the whole derivation process, thus, we should implement two different derivability predicates: one for the last evaluation and one more for the previous *states*. Instead, implementing *ambiguity blocking/propagation* represents better the *state* of each nested derivability predicates invocation and it is more scalable.

5.2 Maggie

In order to clearly describe Maggie-metainterpeter we divide this section into two parts: 1. Defeasible meta-interpreter, 2. Game-Model Meta-Level. This approach is needed since, given an object language (a theory), its evaluation is performed by our defeasible meta-interpreter. As we described above, each time we are trying to evaluate a language we have to move up on a meta-level able to reasoning on the under-layered language. The same rule must be used among meta-levels. Indeed, our game-model is just a particular reasoning on the defeasible meta-interpreter, thus, we provide in the next section a brief introduction to our meta-interpreter and later a description of meta-game-model as to implement it we need to understand how meta-interpreter works.

So far we introduced some features to add into meta-interpreter and we proposed some

syntactic extensions for prolog. Now, we need to refine and deeply describe our implementation.

5.2.1 Defeasible meta-interpreter

Defeasible meta-interpreter must be able to read a theory and try to derive some conclusions according to our previously described principles.

This meta-interpreter has been created starting from a Quintus prolog implementation realized by Nute. Unfortunately, his meta-interpreter had *uncorrected* behavior[13], didn't provide a superiority relation and also its *skeptical semantic* implementation didn't satisfy our needs. So we had to redefine almost all predicates.

Our meta-interpreter can be viewed as human reasoning able to evaluate some basic features of the theory that we put forward. Those basic features can be summarized as follows:

Derivability Given a rule it tries to derive it using the rules in KB. Derivation can be :

- **Strict:** Strict derivation is the same kind of derivation used by prolog engines. It simply tries to evaluate a goal *only* using strict rules.
- **Defeasible:** Defeasible derivation is our extensions and is able to derive using both strict rules and defeasible rules. Since it involves *normal* derivation, we only invoke defeasible derivability of a goal. It's a correct approach because in our OR-predicates *location* strict rules derivability have been written before, thus, it's like if we gave priority to them. In other words, we try to evaluate a goal, *at least* defeasibly, it means that we try to derive it strictly but if the system fails, we try with the defeasible approach.

Using different ways to derive a goal requires a different invocation of itself, so we added a new operator *"@"*. It means that if the goal is preceded by @-operator, we are also

[13] We say that its behavior seems uncorrected since we tested all the typical example provided in its *dpl* file, but we noticed that sometimes rebutting criteria weren't applied as we expected. Maybe, this is due to preemptive predicate in Nute's framework. If you take a look on Maggie code you will notice that we didn't apply ambiguity blocking/propagation criteria in def_der, namely *defeasible derivable*, predicate, but during the rebutting predicate.

interested in its defeasible derivability, otherwise, without, @-operator, we use the typical prolog goal-invocation[14].

Attackability With *attackability* we intend both kinds of attack *rebutting* and *undercutting*. In the previous chapters we widely described the differences between those attacks, and we can simply summarize our previous conclusions:

- *rebutting attacks* propose a new conclusion in opposition of an existing rule.
- *undercutting attacks* just *block* a conclusion provided by a defeasible rule.

Attackability criteria are introduced in the next bullet list, but a wider survey is provided later:

- **Rebutted:** a rule is rebutted if its contrary is strictly derivable. But if a defeasible rule (that is defeasibly derivable), is attacked by another defeasibly derivable rule we implemented two different behavior:
  1. If *ambiguity_blocking* is *disabled* we say that a rule *R* is rebutted by a rule *R1* iff both rules are derivable and there exists a superiority relation in favor of *R1*.
  2. if *ambiguity_blocking* is *enabled* we say that a rule *R* is rebutted by a rule *R1* iff both rules are derivable and there not exists a superiority relation in favor of *R*.
- **Undercutted:** a *defeasible rule* is undercutted by a *defeater* iff are both derivable and there isn't a superiority relation in favor of the former rule.

Decision_criteria Priority relations have been used before as criterion to decide if a rule is to be rebutted/undercutted or not and we introduced them earlier when we talked (more generally) about decision criteria. The other decision criteria we analyzed are: incompatibility[15] and specificity.

[14] Actually, this approach does not provide all the information we could need, indeed from a result we are not able to distinguish which kind of derivation has produced it. Later we introduce a new meta-level that provides this feature using a new operator *$*-operator (on page 72).

[15] Incompatibility rather then be a decision criteria, can be seen as a manner to make a goal undecidable.

- **Incompatibility:** we say that two rules, both derivable, are incompatible if there exists a strict *incompatible/2* rule that argue that, or if there exists a defeasible *incompatible/2* rule whose body is derivable. Incompatibility implementation is used in *contrary/2* predicate. When we try to derive a rule we have to control if there are other defeasible derivable rules with opposite conclusion, but an incompatible rule can be considered as an undercutting criteria. In other words saying: *"incompatible(Rule1,Rule2)"* and Rule1, Rule2,incompatible(Rule1,Rule2) are defeasible derivable, it means that the system cannot reach a conclusion *since Rule2 "attacks" Rule1*.
- **Superiority relations**[16]**:** we say that a Rule1 is superior to another rule, say, Rule2 iff:
  1. there exists a strict superiority rule arguing that.
  2. there exists a strict superiority rule arguing that and its body is derivable*.
  3. there is a superiority rule in favor of Rule1 and there isn't a superiority rule in favor of Rule2 or if exists is not derivable*.
  4. there is a superiority rule in favor of Rule1 and there a superiority rule in favor of Rule2, but those rules are involved in a new superiority relation that argues *non-directly* that Rule1 is superior of Rule2 (and so on...)*.
- **Specification:** In our implementation Specification predicate belongs to superiority relation one. Indeed, if the above criteria fail, we try to see if the condition of the Rule1 is more specific that the condition of the Rule2, in other words if Rule2's Body is derivable using Rule1's Body and not the contrary.

Rejecting criterion:[17] a rule must be rejected when it is the negation of a rule that belongs to our common KB:

- Let R be a Rule such that $R \in CKB$ and R has the shape of:
  - rule*(<something>).*
  - rule(<something>)*:- true.*
  - rule(<something>):= true.

[16]Options marked with star(*) are valid only if *defeasible_priority* option is enabled.
[17]Keep in mind this criterion since it will be of vital importance in the case-studies chapter.

we will reject whatever rule R' such that has the form of:

- neg rule*(*<something>*).*
- neg rule(<something>):- true.
- neg rule(<something>):= true.

where rule can be both a negation rule or a normal rule.

Criteria described above represent the basic defeasible layer of our meta-interpreter and a brief code visualization is provided in the appendix on page 133. What those definition underline is the complex behavior of this implementation depending on mutual multiple invocation among different predicates. As we said above, a rule is derivable if it is not rebutted, to be rebutted there must exist a superiority rule that is derivable...and so on....

5.2.1.1 Tools meta-level

So far, our meta-interpreter is able to answer to some questions concerning evaluation of a goal, but it isn't still complete. We need some tools and some of them can conceptually located in a superior meta-level. As we said in the previous section we would need a new layer able to distinguish if a derivation has been provided from *normal* prolog engine or if the result has been reached using Maggie. Obviously, this component is not properly part of Maggie, but it is a new meta-layer that has to reason on the meta-interpreter.

We defined *$*-operator that is quite similar to @-operator in the sense that tries to evaluate the goal using derivable criteria described above, but distinguishing between strict and defeasible derivation and provides different answers. Furthermore, Maggie provides also two predicates, *whynot* and *say_why_not.* Those predicates proposed by Nute provide a similar result, the former explains why a *goal* is not defeasibly derivable, the other one displays a list of rules and defeaters for a failed goal[18].

One more important rule we need for our meta-interpreter is *dload(+File).* This rule help us to load a file containing defeasible rules, put them in a dictionary data structure that we can manipulate removing some rules or adding more. Actually, that predicate is only used to load a file since the game-model we are developing requires completely different data structure specifications for the rules.

[18]More or less the same difference between *@-operator* and *$-operator*

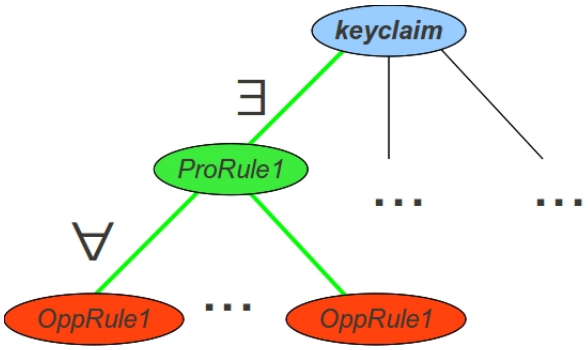


Figure 5.1: OR-AND tree simple representation.

5.2.2 Game-model meta-level

In the last section we introduced Maggie[19] meta-interpreter, now we are going to show a possible implementation of a specific game-model defined in [38]. Our aim is to create a recursive game-model that involves two parties in an argumentative debate. We represent its evaluation process using an OR-AND tree[20], where the root is the keyclaim, and its children are all the possible moves that the opponent can put forward. With respect to figure 5.1, if there exists (OR) at least one branch in which the proponent wins in the next step for each (AND) of its moves, the debate is won by him.

Thus, each node of the tree represents a new state characterized by the fact that a new rule has been put forward. Since, our meta-interpreter is based on a non-monotonic logic, it means that evaluating the derivability of the keyclaim in different nodes the result could change.

In every nodes of that tree we should evaluate the keyclaim asserting a related set of rules that are valid just for that node. Indeed, when one of the players puts forward a new rule, Maggie will have to remember which of the player has moved and in which node he did that move, thus, it will evaluate a different possible evolution represented by a unique branch.

In figure 5.2 we show the distribution of the different knowledge bases involved into project; private ones are only known by related owners, instead, the common knowledge is public and accessible by both the parties. Maggie represents an intermediate layer that

[19]Pay attention, from now saying Maggie-MI we only refer to the meta-interpreter, instead, saying Maggie we refer to whole prolog application containing Maggie-MI, tool and game-model implementation.

[20]In literature are also called Exist/For-all.

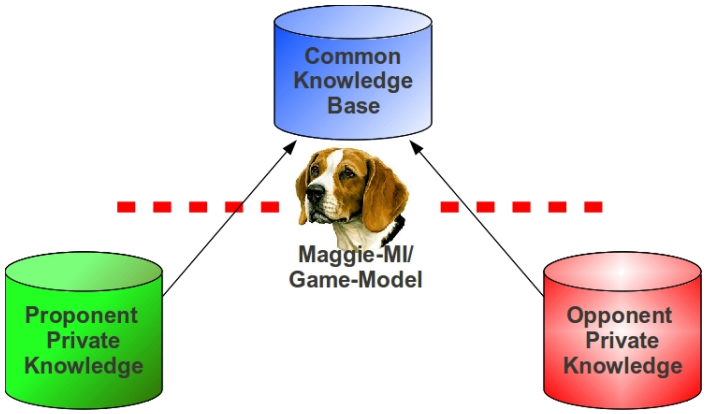


Figure 5.2: Knowledge repository and Maggie evaluation

takes in turn a rule from each private KB and evaluate the keyclaim. So it is like a judge that, evaluating the common laws (rules) and the arguments putted forward by the lawyers (players), pronounces sentence.

The implementation is based on the pseudo-code defined in [32] (chapter 10) and it is made up of two fundamental predicates:

won(proponent): proponent wins the debate if:

1. Keyclaim is strictly derivable;
2. Keyclaim is defeasibly derivable and the *opponent* exhausted his moves;
3. Proponent has got a valid move and using that move in the next step, opponent will lose.

lost(opponent): opponent loses the debate if:

1. Keyclaim is strictly derivable.
2. Opponent exhausted his moves and keyclaim is still defeasible.
3. Proponent will win in the next step and so far negation of the keyclaim is not derivable.
4. Proponent will not win in the next step and the negation of the keyclaim is derivable.

won(opponent): opponent wins if:

Proponent loses if negation of keyclaim is derivable	vs	Proponent loses if keyclaim is not derivable

Table 5.1: Logical proponent lost criteria.

1. Negation of the keyclaim is strictly derivable;
2. Proponent exhausted his moves and negation of the keyclaim is derivable;
3. If he has got more moves to put forward and in the next step and Proponent will lost.

lost(proponent): proponent loses

1. Negation of the keyclaim is strictly derivable;
2. Proponent exhausted his moves and keyclaim is not derivable;
3. Opponent doesn't win and keyclaim is not derivable;
4. Opponent doesn't win and keyclaim is derivable the negation of the keyclaim;

Unluckily, although this approach is really elegant we will see some complexity problem related to a possible double deliberation in lost predicate (point 3,4). Furthermore, it should be noted that during valuation we can assume different criteria in order to determine if proponent has won.

This remark can be considered the point at issue since it involves the first we presented. Indeed we need to use two different options in lost predicate because this unclear definition of lost criteria. In this case we assume two different approach both applicable:

1. We use both point 3 and 4 of lost predicate accepting the risk of a double evaluation.
2. We just use negation of the keyclaim and when it's needed we could switch to a skeptical semantic in order to avoid that both *keyclaim* and its negation could be true.

In this thesis we used both approaches and in complexity section we will reach some conclusions useful to get better performance.

Implementation of a meta-game-model requires also a data structure that can be understood by humans, in particular a tree derivation tree able to represent all possible path of

our debate. Tree is built starting from the root and recursively goes down till the leaf nodes requiring in backward the generated data, it means that in a given node, after won/lost predicate invocation, prolog cannot remove from the stack the invoking predicate and it must remain open and wait till the end of the generated sub-tree explorations. Since each node represents the addition of a new rule by one of the parties, we have a node for each move, and for each branch that rule is putted forward in a different levels.

Actually, the trees that we have to explore are two, the first one tries to reach a victory of the proponent, the other one a victory of the opponent; but, since their behavior is rather similar, in the appendix we will show just the first one (see on page 135).

The game starts from a keyclaim attended by first proponent move, that can be considered wrong since keyclaim sounds like a proponent rule, but it isn't for a simple reason: after expressing the keyclaim, proponent has the burden to defend it and try to prove its truth.

5.3 Basic protocol

In this section we show the basic protocol of the game-player explained in [32] and that we present in an algorithmic representation.

To initialize and execute the game-model we apply Algorithm 1: system loads all the rules and builds Proponent won tree. In this case we use the protocols described in Algorithm 2 and Algorithm 3. In Algorithm 2 we show in detail the behavior of game-model from the proponent's won criteria point of view. We can distinguish it into three possible cases of victory:

1. Keyclaim is strict;

2. Keyclaim is defeasible and opponent doesn't have valid rules to put forward;

3. We put forward a new Proponent's rule invoking *lost* predicate.

Note, that third point requires that each invocation of *lost* predicate has to succeedes; this behavior corresponds to AND part of OR/AND tree.

Algorithm 3 shows the behavior of the other part of the game-model. It's quite similar to algorithm 2 but in this case we are not seeking if proponent won, but if opponent lost. It's

Algorithm 1 Game-model's initialization and starting goals execution.

```
begin
  load data structure for Opponent and Proponent repositories
  load data structure for Proponent and Proponent repositories
  load configuration settings such as defeasible priority, ambiguity propagation etc...
  load Keyclaim
    if won(pro,CKB,ProKB,OppKB,OldTree,Newtree) returns true
      write NewTree in tree.txt file
      return true
    else if won(opp,CKB,ProKB,OppKB,OldTree,Newtree) returns true
      write NewTree in tree.txt file
      return true
    else if return draw
  end
```

necessary since we need to use a different predicate to put forward rules from opponent's private repository. Also in this case we underline three opponent cases of lost:

1. Keyclaim is strict;

2. Keyclaim is defeasible and opponent doesn't have valid rules to put forward;

3. We put forward a new Opponent's rule invoking *won* predicate.

Note, that third point does not require that each invocation of *lost* predicate has to succeedes; this behavior corresponds to OR part of OR/AND tree.

Those Algorithms shed light on the recursive approach, in particular executing *won/6* predicate and reaching the third *"if option"* we invoke *lost/6* predicate, and it does the same. That is the reason why we state that this recursion is *mutual.* Furthermore, both third *"if options"* have to invoke as many times as the number of valid rules in OppKB/ProKB. That is the reason why we state that this recursion is also *multiple.*

If pseudo-code we illustrated fails, system will use two rather similar algorithms in which a success means Opponent won. Since this second tree derivation is the dual approach and it is really similar to the one we described, we decided to omit it, but if you are interested, Maggie code can be consulted on:

http://splogad.altervista.org/target/site/maggie/source/defeasibleMI.pl.txt

Algorithm 2 Game-model protocol: won(pro,CKB,ProKB,OppKB,OldTree,Newtree).
Tree exploration of the debate between proponent and opponent.

Input: $X \subseteq$ *pro, CKB, ProKB, OppKB, OldTree.*
Output: $Y \subseteq$ *NewTree.*

```
begin
  assert all rules into CKB
    if Keyclaim is strictly derivable
      create NewTree starting by OldTree
      return NewTree
    end
  retract all the rules

  assert all rules into CKB
    if Keyclaim is defeasibly derivable && Opponent exhausts valid rules
      create NewTree starting by OldTree
      return NewTree
    end
  retract all the rules

  for each rule ∈ ProKB do
      if the rule is valid
      insert rule in CKB
      delete rule from ProKB
      invoke lost(opp, NewCKB, NewProKB, OppKB, OldTree, NewTree)
  end
  if each valid rule in ProKB returns true from lost/6 predicate
      create NewTree starting by OldTree
      return NewTree
  end
end
```

Algorithm 3 Game-model protocol: lost(opp,CKB,ProKB,OppKB,OldTree,Newtree).
Tree exploration of the debate between proponent and opponent

Input: $X \subseteq$ *opp, CKB, ProKB, OppKB, OldTree.*
Output: $Y \subseteq$ *NewTree.*

```
begin
  assert all rules into CKB
    if Keyclaim is strictly derivable
      create NewTree starting by OldTree
      return NewTree
    end
  retract all the rules

  assert all rules into CKB
    if Keyclaim is defeasibly derivable && Opponent exhausts valid rules
      create NewTree starting by OldTree
      return NewTree
    end
  retract all the rules

  for each rule ∈ OppKB do
      if the rule is valid
      insert rule in CKB
      delete rule from OppKB
      invoke won(pro, NewCKB, ProKB, NewOppKB, OldTree, NewTree)
  end
  return NewTree
end
```

5.4 Complexity

In this section we discuss some performance tests of our system and we try to propose some suggestions in order to get better tree representation and evolve the system for future improvements. First, we present some evaluation related to resources of time and memory, later we will show you some attempts to improve performance.

We tested the application adding some theory with a growing number of rules, on a Centrino 2 T2300@1.66Ghz, 1Gb Ram, kernel Linux 2.6.32.22-generic. Our assessment provides some information about the time (measured in ms) to finish the game, and the total amount of memory used during the task execution. At the beginning we were not able to get a result (except for trivial examples) since stack limitation[21] blocked our application, and system during calculation was becoming slowly till arrive to deadlock; fortunately, we solve that problem since on Linux is possible to change the stack limitation to unlimited size. One more problem that we underline is CPU usage that we didn't include in this evaluation since the application is not optimized for working with more processors and multi-threading.

Looking on the chart of Figure 5.3, in the (a) item we noticed in 4x4 example an excessive usage of memory, instead time evaluation is about 2.5 seconds. It is not a great result, but considering that application is not yet optimized we could accept it. Resources usage becomes almost *out of control* just adding one more rule for each party. In the chart (b) we can see that time needed for evaluation is 4.079.490 ms (about 1h and 8 minutes), and the task requires more than 69MB.

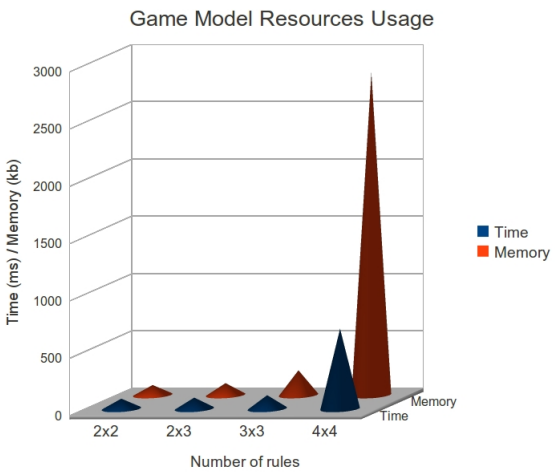
This result is not encouraging because a real theory should contain a great number of possible rules to put forward and since our application is really slow also to compute a minimum number of rules a deep analysis it's needed to fix those problems.

There are two ways we can use to analyze this problem:

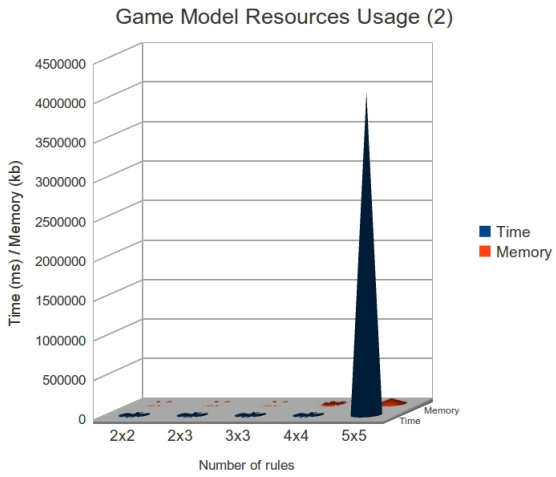
1. Improve our programming approach;
2. Analyze our formal definition and check if there are some theoretical errors;

The first approach is less interesting and could be applied in any case (see 5.8). The second one is more appealing since it involves complexity study, logical analysis useful to provide some suggestions for the future works.

[21] On Linux stack is fixed to 8192kb for each shell.



(a) With at most 4 rules for each of the parties.



(b) With 5 rules for each of the parties compared with previous results.

Figure 5.3: Game model resources usage.

To better understand the causes of this huge increase of time and memory we measure the complexity of the system:

State_Space_Complexity: with this index we indicate the number of reachable positions from the init state. It means to evaluate all the possible state we reach including intermediate nodes:

Given K and Q, where K is the proponent number of rules and Q opponent number of rules, the number of total nodes would be the following succession:

$$nodenumber = 1 + k + k * q + k * q * (k-1) + k * q * (k-1) * (q-1) + k * q * (k-1) * (q-1) * (k-2) + k * q * (k-1) * (q-1) * (k-2) * (q-2) + ...$$

that could be express with a recursive function. Nevertheless, our aim is to concretely see a quick complexity size, so we can round down this number using only depth and branch factor in this case we use a depth (d) of k+q (in the worst case) and a branching factor (b) that is the average among k,q.

In this case we have a complexity about $O(b^d)$;

Game_Tree_Size: game tree size represent the number of conclusion that we can reach, in other words the number of child nodes of our tree, namely, $k! * q!$;

Game_Tree_Complexity: the minimum number of leafs that we have to explore coincide to the total number of leafs, namely, $k! * q!$;

Computational_Complexity: computational complexity depends on number of input arguments, we can see that the number of items increases factorially (that is worst than exponential), furthermore, for each node instance we would create a data structure containing three different lists (Common KB, Opponent private KB, Proponent private KB). It should be noted that in a 5X5 rules debate we would have 14400 leafs nodes each containing a different data structure whose size is between 1Kb-3Kb (in our simple examples). It means that just for node children we would need of about 50MB of memory. It is easy to demonstrate that this problem is an ***ExpTime*** problem and also about memory we have a exponential complexity.

Thus, it would be better analyze our approach before attempt to optimize the code.

5.5 God, Complexity and Acid Cut

Why tree exploration can lead system to slowing down? Leaving aside mathematical problem and informatic tree derivation issues, logically, we are trying to simulate human reasoning with a logic programming, and we are working just on a logical approach, maybe forgetting what is the real human approach.

Roughly speaking, humans don't create trees, nor any similar data structure. If we remark the typical reasoning approach, we notice that we are used to use almost always a depth first approach. Let suppose the reader is not convinced, why don't you try to draw a tree on a paper, we mean a tree like the ones we explained above with three rules for each of the parties; you will notice that A4-paper are really small and that all the branches are almost useless. That's normal since we reason creating branches only when there is the evidence that those branches are gains for our purpose. The problem is that we are able to reasoning, and learn over the year when is the case to create some branches. But, sometimes we are wrong since we tried to reduce complexity despite our strategies acquired over the years. This section title involves the word God, since existence of God is the typical acid cut that we perform to avoid a complex (almost paradoxical) reasoning[22]. Indeed, there are

[22]Take a look on the notable people who tried to prove the existence of God as for example Göedel. He

two kinds of people: who believe in God, and the ones who reject its existence. It's funny noticing that both are wrong. If human brain was written in prolog, we could summarize those beliefs using a few lines of code:

- People who believe:

```
exists(god, john) := has_faith(john).
```

- People who don't believe in God:

```
neg exists(god, splogad) := \+ proved_existence(splogad).
```

Both rules are defeasible, but talking with both kinds of people they seem strict rules. It's like if those people tried to change the strength of the rule. Their reasoning present a deep lack of proofs, the first one demonstrates God's existence using the faith, in other words, without demonstrating its existence. The others will change idea only when someone will prove God's existence. So it's like if in their mind those rules were strict.

Dynamic change of the strength of a rule it's not a novelty in literature, instead, using this approach to reduce complexity it could be. Indeed, we suggest for future works to develop a meta-level able to change a rule from defeasible to strict when needed or according some principles that we explain below and the following definition:

Definition 1. *With Acid Cuts we mean a prunning criterion able to reduce the size of the tree and avoid the excessive growth of numbers of nodes during the exploration tree. According appropriate principles (see below), it is able to dinamically change the strength of a rule and keep on evaluating the keyclaim derivability "with gambling".*

Principles:

- Time constraints (if a rule is derivable during a pre-defined time, we can consider it as strict);

passed a lot of years studying that problem or better trying to defeat all the scholars theories that preceded him and that were accepted as philosophical demonstrations. Göedel lost a lot of time achieving his goal and he dead after some months of madness during which he was sure that someone wanted kill him poisoning his food, so he stopped to eat but meeting death. It could be a nice demonstration of how it is complex and paradoxical logic field and that existing of god is just superficially analyzed by normal people.

- Defeaters disabled (if all the defeaters that attack a goal are false since one of the term of their condition is strictly false, we can assume that the defeasible rule will never defeated by them);

- More than one of the terms of rule's condition is strict (we can assume that the whole rule could become strict).

We demonstrate that changing the keyclaim strength in a certain part of the debate, the complexity of the program will be reduced, indeed, we reached a quick result with four rules for each party without changing stack size.

We call this kind of cut, Acid cut, since they are not completely true, but can reduce the amount of memory used by the task without introduce inconsistencies (if well developed). An acid cut could be interesting also to analyze the strategies, we mean that is like to hop up the system and can be interesting see the debate prosecution after a *cynical / not completely correct* alteration of our theory. What could happen if one of the party strictly assert something that is not true, and opponent is not able to change or notice that change? Could the worst lawyer win the debate cheating?

Why reduce complexity with gambling? Actually, during writing thesis we tried to propose as much suggestions as possible to fix performance issues of our system. Nevertheless, this approach is interesting also to increase similarity between artificial intelligence and human reasoning, indeed, we can assume that humans don't reach conclusions always reasoning in the same way, they usually answer to a question quickly using the simplest way of reasoning that they could use. That is the reason why sometimes we listen to people saying sentences such as "*I'm sorry. I answered hastily!*" or "*I've been racking my brains all morning trying to think of an answer to your question!*". Those two sentence shed light on a probably multi-level reasoning approach of humans, say, that the first sentence could correspond to an *Acid Cut* and that was not appropriate in that context, so, after a feedback from interlocutor that make us aware of a mistake in our previous conclusion, we have to reply saying "I'm sorry, I'm wrong", namely, reasoning on that problem takes more time than expected. In the second sentence we can see a problem rather similar to our complexity issues, namely, considerable amount of time spent to evaluate a conclusion, and after al the morning we are still waiting for a conclusion. It means that we can approximate human reasoning to a multi layered system in which each level is able to cope the problem with a different deepness.

5.6 Tree analysis

One may ask :"Is it necessary to build this tree?". Indeed, this tree is not a suitable way to analyze all the possible cases. Generally, two possible aims require building a tree:

1. Is there at least one possibility of won using my theory? (CASE1)
2. Is it sure that i will win the debate using this theory? (CASE2)

Also if those questions seems rather similar, they aren't. Roughly speaking, this approach represent a strategy tree in the case that proponent can only win. But this isn't a general case. Usually, we use a theory in a wrong way and maybe using a different orders of the sentences we could change the result. Unfortunately, this tree represents limitation when:

- We provide a dynamical changing of the strength of a rule;
- If the number of rules in theory proposed by two parties is different.

Let's first analyze the second item. Indeed, if we suppose that Proponent has got two[23] more rules than Opponent, and one of them is a MatchPoint (we use this name to define a rule that surely bring one of the party to a won), there will be some branches in which the proponent won't be able to put forward his MatchPoint and he will lose, instead in the other branches he will win. Looking at figure 5.4, we reach the first two leafs. Let's suppose that in the former leaf node the keyclaim is not derivable, proponent would lose, but in the next leaf node he would win as it will use the MatchPoint rule. In this case the system fails both tree evaluations and it returns "It's a draw" and it is not correct. So, if we are in CASE2 this approach could be acceptable, but we cannot consider it as a rule of thumb. (The same would happen if we add cutting criteria caused by a dynamical changing of the strength of a rule.

Thus, it means that this approach, we mean OR/AND tree, is not always acceptable, and it would be better using a OR/OR tree.

Definition 2. *We say that a theory is* **persistent** *if it is not subject to dynamic change of strength of the rules, if all the rules that belong to the theory cannot be rejected and if the*

[23]We refer to general case of *two more rules than* the other payer, because it depends also on who is the first player putting forward a rule. Indeed, to get an ambiguous tree derivation in the first tree, we need just one more rule for proponent only if opponent starts, otherwise, two ore more rules.

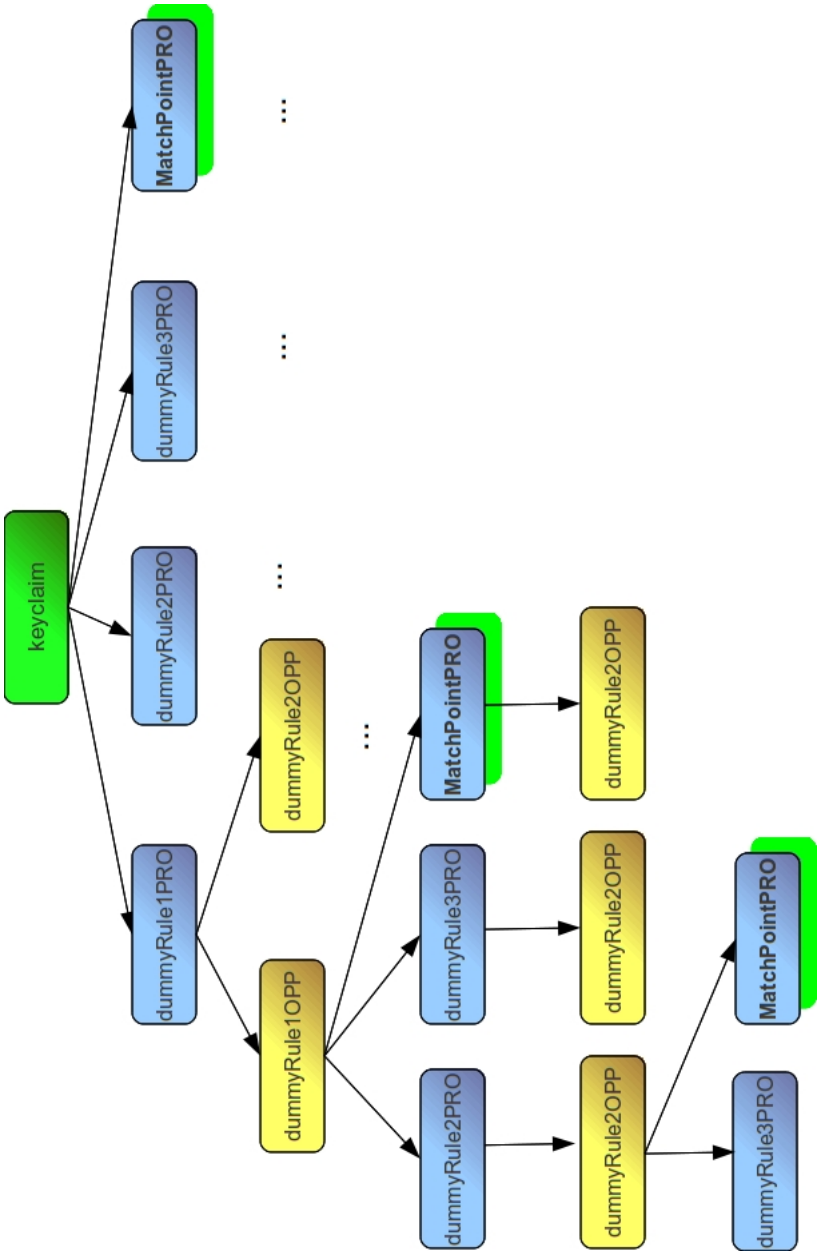


Figure 5.4: Ambiguous derivation OR/AND tree.

players have got the same number of rules to put forward or at most one more rule for the player who goes first.

Theorem 3. *OR/AND tree can be used only if the theory is persistent.*

Proof. A player won if there exists at least one branch in which all its children lost, it means, that if we notice one or more cases where we cannot use OR/AND tree as a rule of thumb, we must use OR/OR tree derivation.

Let suppose T be a **not** persistent theory, then during its derivation process we could reach unforeseeable final states since permutations could be different and change in a different way since we have different context.

Let K be the number of rules that Proponent can put forward and Q Opponent's rules number such that $K > Q$ and that both K and Q can allow $MatchPoint$ rules. Then there could exists at least one permutation that allows a different result. Let $defder(CKB(P_1), Key)$ (where Key is the keyclaim and CKB contains only a subset $SK$of Proponent's rules and all Opponent's rules) be not true, if $K - SK$ is the set of Proponent's rules that we didn't use before, and that could contain a MatchPoint rule, than in the next branch in which MatchPoint rule will be used, and makes $defder(CKB(P_i), Key)$ true and reach a different conclusion than before.

Case of Acid cuts

A **not** persistent theory could contain ***Acid cuts***, different number of rules proposed by players, or rules that can be rejected. In the first case, we can alter a rule in a given context and prune the tree if there are the conditions to change the strength of a given rule $R \in ProKB$. Depending on the contexts we can have the following states:

$R \in SK$ and it remains defeasibly derivable;

$R \in SK$ and it becomes strict;

Both cases are possible and could bring the system to reach a different conclusions, namely, same rules but in a different permutation can reach different conclusion in the final step.

Case of rejected rules

If a rule is rejected in a given context, it means that in another context it is allowed and the system can reach a different conclusion. Notice that rejecting a rule we alter number of available rules, it means that survey next case is needed.

Case of different number of rules in private repositories

This is the most trivial case since if we have $K > Q + 1$ it should be noted that permutations reach different final steps caused by the fact that they use different rules. □

Unluckily, Theorem 2 says that OR/AND tree would never be applied since if we have the same number of rules, in each leaf node we always assert the same rules and so if we noticed that in the first branch proponent won/lost, he will win/lose in every leaf nodes. In this case, we could drastically reduce the complexity of our tree using a depth-first approach, namely, from $O(k!q!)$ to $O(k + q)$.

Theorem 4. *If a theory is persistent, then depth first-search is enough to know who is the winner.*

Proof. Let two players Proponent and Opponent have got respectively K and Q rules and K and Q are such that:

K = Q , or,

K = Q+1 iff Proponent will go first, or,

Q = K+1 iff Opponent will go first;

and that the keyclaim is Key.

Let suppose that *Acid-cuts* are not allowed, that there are no rules that could be rejected and that derivability is given applying the following formula:

$Kder(i) = defder(CKB_i, Key)$. *where* i *represents the* i-th *step during the evaluation.*

and that we are only interested in the final step. So we replace *i* with *K+Q*.

All the possible available branches in this derivation task are given by the permutation of that rules, and since the theory is persistent, all the branches in case of won reach the leaf node and they are represented by all the possible combination of the rules that are $K!Q!$. Now we define all those leaf nodes as a possible result of the permutation of the two sets of rules provided by the players.

Thus, we have that:

CKB(P_1) is the final step of the first permutation and it contains all the rules;

CKB(P_2) is the final step of the first permutation and it contains all the rules;

...

CKB($P_{K!Q!}$) is the final step of the first permutation and it contains all the rules;

Since all the final steps contain all the rules we can say that intersection of all CKB(P_i) is the set containing all the rules provided by player, say, CKB_g.

But if :

$CKB(P_1) = CKB(P_2) = ... = CKB(P_{K!Q!}) = CKB_g$,

then

$Kder(Q + K) = defder(CKB(P_1), Key) = defder(CKB(P_2), Key) = ... = defder(CKB(P_{K!Q!}), Key) = defder(CKB_g, Key)$

So any branch is able to give the result. □

If theory is not persistent, we simply use an OR/OR tree[24].

5.6.1 Adding rules in Maggie

In previous sections we noticed that system's performance is compromised since intrinsic recursive structure of game-model and also by the fact that we are putting forward new rules from private repositories randomly, namely, without reasoning on their meanings.

What does it mean reasoning on rule's meaning? Simply, we want to put forward a rule only if it is the case to use it, in other words, only if a player can approach to achieving goal using it. Thus, we need to define and formalize when is the case that a rule can be asserted by a player in order to avoid the assertion of "*useless*" rules that slows down the system since they aren't needed in a given context.

Let suppose that in the future a software like the one presented in this thesis will be used in legal domain. It could be possible to load into the program a great number of rules and some of them, in a given instance, it is not the case to evaluate. We mean that if we are dealing with a case of murder, it would be better if the system, or better, one of the player doesn't use *Highway code's rules* just to give the impression to have more rules. Indeed, using too much rules not only requires more resources in term of space and time to reach a conclusions, but also we are losing touch with our aims, we mean an *intelligent system* and not a random rules evaluator.

The problem of reasoning on a rule is that it could depend on *human language comprehension* since we need to understand not only the meaning of a rule, but also if asserting it can be usefull to change (to take advantage) the meaning of the whole rules evaluation. In this case how can we understand if a rule changes keyclaim derivability result?

[24]So don't worry if using Ubongo-0.5 we see a pruned tree, since it is a normal behavior: if the opponent doesn't win in a branch, it won't build that branch.

Rule *R* is **not** recommended	Rule *R* is recommended
R is not the case to be introduced, but if it is the only rules in private repositories it could be asserted	*R* can be used.
***R* for now is rejected, but it could be used later so we leave it in private repository**	
R is to be removed from private repository	

Table 5.2: Reasoning on rules: different available criteria.

We already found a potential problem, we mean our reasoning it will be just a suggestion or could be also a rejecting criteria? Namely, if we notice that a rule is not the case to be put forward we can label it as not-recommended or reject it. In the latter case we cannot remove it from private repositories, since it could depend on one more rule not yet asserted and could be convenient store it for future evaluation (see Table5.2).

What does it mean to suggest a rule? We said that we can use a rule only if it is the case to use it, in other words if it is convenient to put forward a rule for the player who is currently playing.

To better understand how to implement in Maggie this approach we need to formalize a set of recommendation rule criteria for both proponent and opponent:

"A basic protocol for the admissible moves of the players could be, for the proponent, that the current move attacks the previous move of the opponent, and that the main claim (the content of the dispute) follows from the arguments assessed as currently valid."

"For the opponent we have that the arguments of the move attack the previous move, and the main claim is not derivable."[38]

The text quoted above explains, verbally, criteria that we would like to formalize. Actually, in previous section we already said that one of the goals of each player is to reach respectively *keyclaim derivability* and *keyclaim non-derivability* depending on *Proponent* and *Opponent.* The novelty of the approach we quoted above is a mutual goals dependency among the parties, indeed, they not only have to achieve the intended *keyclaim* status, but also create a debate using an *intelligent way* in the sense that we cannot allow system to *randomly* put forward all rules contained in private repositories, but reasoning on both *keyclaim* and last *enemy's rule.*

We represented what we stated above in Algorithm 4[25], where we suppose that opponent goes first and attacking keyclaim means put forward:

1. A rule (defeasible or defeater) whose conclusion is the negation of keyclaim;
2. A rule that is a term of the condition of a rule whose head is the negation of keyclaim.

Note that this approach is an improvement since we apply a well-advised and *a priori* cutting criterion able to reduce complexity without make system non-deterministic. Nevertheless, our definition is quite elementary, in the sense that we omitted some important rules we define throughout writing the thesis such as *incompatibility* and *superiority_relations.*

In that case we suggest to use the same algorithm for both proponent and opponent. If a given rule fails previous putting forward criteria and has the form of `sup_rule(X,Y).` or `incompatible(X,Y).` and is able to make keyclaim respectively *defeasible* for proponent and *non-defeasible* for opponent, then is the case to put it forward.

We tested this behavior in Maggie-Beta1-Test1 and we achieve our aims in term both of performance and *quality*[26].

Furthermore deserve a mention possibility to add more than one rule in each turn. Indeed, sometimes, rules are not independent and need the support of other rules to be considered by the system Recommended.

> **Example.** Let suppose Proponent argues a *keyclaim* and that Opponent can attack *keyclaim* just asserting one rules, if we need more rules to argue an attack, we have to side effects: we risk to generate too much rules, we could ignore some rule that in a given step aren't Recommended but in a context in which other rules are involved it becomes usable. The latter problem could be serious if we accepted criteria of Table 5.2 line 3.
>
> *So let Keyclaim be:*
>
> ***keyclaim.***
>
> *and Opponent have got the following rules:*

[25] In this algorithm we suppose that Opponent starts the game, it is quite similar to the case in which starts Proponent. We provided different implementations of Maggie for testing the game using both criteria: opponent goes first vs proponent goes first.

[26] With quality we remark what we stated above, namely, the need of a system able to put forward rules in a reasoned way and not just randomly.

Algorithm 4 Rule to put forward selection.

begin
load *keyclaim*
*/*let suppose that keyclaim is derivable at the beginning*
and that Opponent starts the game/*
Opponent:
if *there exists at least one rule whose head is the negation of* ***keyclaim***
and it is not derivable but we have a rule R that is a term of its condition
*that can make the first rule derivable (****Keyclaim*** *attack)*
put forward *R*
end
read last rule
if *there exists at least one rule whose head is the negation of last added*
proponenent's rule, more specific and it is not derivable but we have a rule R
that is a term of its condition that can make the first rule derivable
put forward *R*
end
Proponent:
read last rule
if *there exists at least one rule whose head is the negation of last*
opponent's rule, more specific and it is not derivable but we have a rule R
that is a term of its condition that can make the first rule derivable
put forward *R*
end
read last opponent rule
if *there exists at least one rule whose head is the support* ***keyclaim***
and it is not derivable but we have a rule R that is a term of its condition
that can make the first rule derivable
put forward *R*
end
end

1. **neg keyclaim := term1, term2.**
2. **term1 := term3.**
3. **term3.**
4. **term2.**

If in a first sight we should consider only rule 1 Recommended. After asserting that rule we would need to assert in the following steps the assertion of the rules that prove the condition of rule 1.

In case explained above, we would need to create more data-structure, build more tree if we allow just one rule to put forward in each node. Instead, if we allow opponent to put forward not only rule 1 but also all the rules that support it, namely, 2,3,4 we could get a smaller computational complexity since we reduce tree size and it would be more similar to human reasoning.

5.7 Ubongo

Finally, we show the graphical interface developed to easy use Maggie. As we said above we called it Ubongo and it consist of three simple components:

- Engine (The core of the application that has the job to manage the other components);
- Parser (An extended Prolog parser that accepts also defeasible rules and new kind of rule to configure the challenge);
- MI (Maggie);
- TreeViewer (A simple viewer of the derivation tree);

Project has been developed using Java(TM) SE Runtime Environment (build 1.6.0_20-b02), eclipse Galileo and javacc 5.0. Application is just the graphical interface front-end that provides some example by default, but it is also a nice text-editor.

It simply take the theory from the text viewer and send it to the parser that has the job of evaluating the syntax. It is a prolog parser built over YProlog and it has been modified since it was no able to evaluate *defeater, defeasible rules* and *"rules"*.

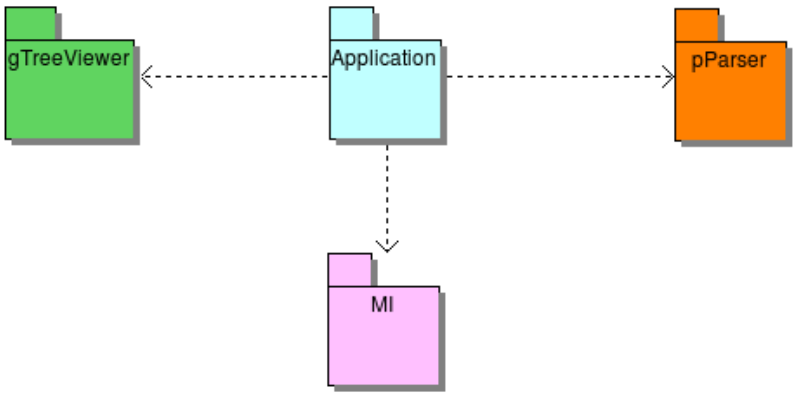
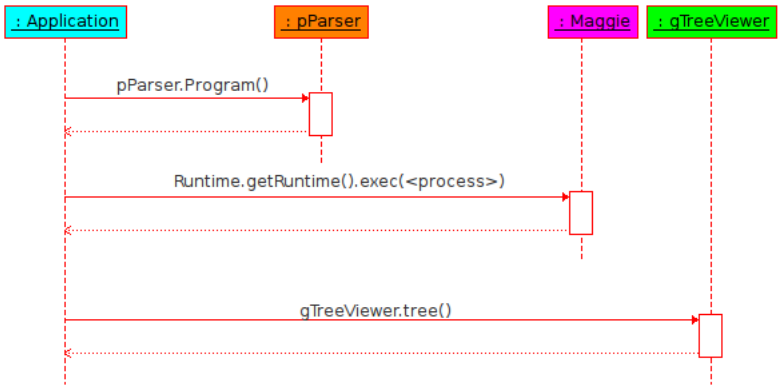


Figure 5.5: Ubongo packages diagram.

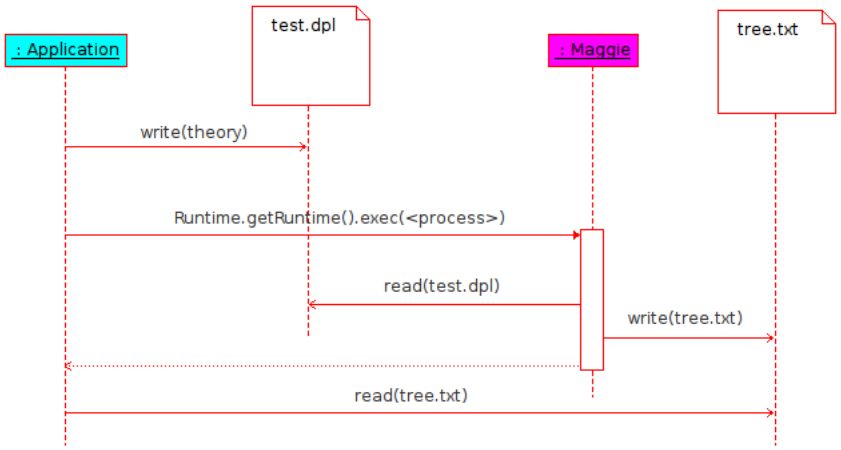
With “rules” we mean some rules that we added in order to make possible the initialization of the challenge, and all available rule are:

- rule(key, <keyclaim>) : is the rule that we have to use in order to put forward the keyclaim;
- rule(pro, <rule>) : <rule> is a rule that belongs to Proponent private KB;
- rule(opp, <rule>) : <rule> is a rule that belongs to Opponent private KB;
- rule(ambiguity_blocking, <on/off>) : enables or disables the ambiguity propagation;
- rule(defeasible_priority, <on/off>) : enables or disables defeasible priority;
- rule(orandTree, <on/off>) : enables or disables Or/And tree derivation;
- rule(metaSupRel, <on/off>) : enables or disables the meta superiority relation level to control possible paradoxes putted forward in CKB;
- rule(mess, <rule>) : is a rule that we can use to add some message rule as for example write(’<text-to-write>’);

It should be noted that it becomes easy writing a theory in this way, and clicking on the execution button (after the parser evaluation) Ubongo tries to access to a script and execute it from java. This task is performed opening a command editor (shell in Linux) whose stack size is increased to unlimited value. The external usage of Maggie allow us to simply add



(a) Complete theory execution diagram



(b) Detailed sequence diagram between application and Maggie

Figure 5.6: Theory execution sequence diagram.

a folder in Ubongo's root directory in a completely modular scenario. If you want to use Maggie 0.7 or a customized version of Maggie instead the last Beta release, you have just to change the folder.

Actually, there isn't a direct invocation from java to prolog, since our aim is to leave this application non Sicstus-dependent, so just changing the path of the prolog engine a new script for prolog invocation will be built and executed.

5.8 Future works

In this chapter we discuss some suggestions for future works and we also want to clear up some alternative approach in order to improve Maggie performance. During previous sections we focused on tree analysis and we hope to have provided enough and deep informations that can help to improve this system. But we also mentioned a different way as a solution of system slowing down, namely, code optimization.

Due to lack of time, we cannot develop more programming approaches for our application, but we can suggest some interesting way to speed up Maggie engine. First, we have to find our bottleneck and assess if we can fix it.

As we saw in the game model code, slowness is due by recursion since it is mutual and multiple. It seems really difficult to apply a tail recursion in this scenario, but we may change something in our algorithm and prevent that recursion fill the stack or at most prevent an excessive processor usage.

Taking a look on the code, we notice that *find-all* predicate, not only cause a problem since, also if it was the last rule in the clause, it requires multiple invocation of the same predicate, thus, tail recursion can be only partially implemented; but it cannot be the last rule because after find-all execution we need to perform some check on the result in the case of OR/AND tree and evaluate if won/lost predicate is verified for all child nodes.

To fix this implementation we can completely change our recursive approach and edit our algorithm about:

- forward tree building;
- find-all parallelism;
- delaying the OR/AND check;

The first item indicate one of the heaviest recursive problem, namely, a parent node wait until the children sub-executions end take their resulting sub-lists and build its sub-list. To perform this task we have to reserve to much memory and make processor task slows down. Thus, it would be a better solution if parent node send its “up-list” to its child nodes, and also if the tail-recursion practically cannot completely close parent predicate, we can save up memory since allocation decreases. But how many time it takes the complete execution of its last invocation, namely, the find-all predicate? Actually, it is a thorny

problem since multiple recursion is not easy to be prevented, but it should be noted that the nested execution of find-all arguments is completely independent. It means that we can execute its argument in parallel exploiting Muse framework. This framework, provided by SicstusProlog, leaves unchanged the prolog syntax and just asserting a new predicate: "`muse_flag(num_workers,_,<number_of_workers>)`" we could improve our performance. From sicstus web-page there are some examples that report a speed up about 3.8; in our case of 5X5 rules, it means a time-saving from 1h8' to 17'. It's a great result, but we have to mention that time keep on increasing exponentially, and 5X5 example is just an elementary example. Nevertheless, combining all the changes, maybe we could achieve an even better result. Finally, we said that find-all predicate cannot be the last rule since we need to check if at least (OR) one branch wins and for all branches (AND) opponent loses. That is not a problem, indeed, if all the child nodes before ending make tracks of their list (note: now the list is forwarded) in a data structure, we are able to build the tree in bottom-up.

Furthermore, there are more solutions such as a complete rewriting of the code in a strongly typed language. In particular we suggest Erlang. This project doesn't need a presentation, but deserve a mention. Since 2001 it has been integrated in Ericsson's Open Source Erlang/OTP system; it *aimed at efficiently implementing concurrent programming systems using message-passing in general and the concurrent functional language Erlang in particular,* namely, it has been invented in order to improve recursion performance.

We guess that those suggestions are enough to provide a nice starting point for a future work based on this thesis survey.

5.8.1 Multiple keyclaim approach

What we did is not enough since we are dealing with logic programming and reasoning on computational features we risk to lose touch with another aim we would like to reach, we mean human reasoning simulation. Indeed, a limitation of our approach is caused by the fact that we are using just one keyclaim, but, usually, we would need more than one.

Let suppose that we want to evaluate if a given strategy (or better using a given theory) can help a lawyer to demonstrate that the suspect, say John, is not guilty. Probably, investigations proved that he is guilty, so using a keyclaim such as *neg guilty(john)* would be useless. Imagine that our system is able to notice that with a particular theory will lose without

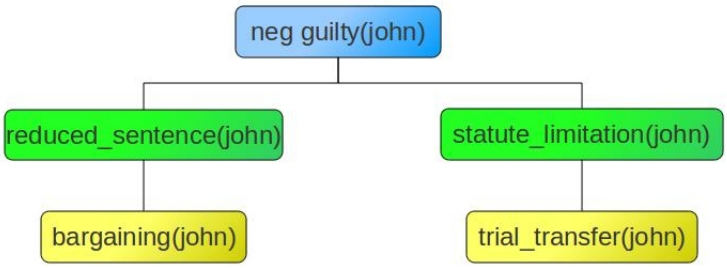


Figure 5.7: Multiple keyclaim tree derivation.

a doubt, but using a different keyclaim we would reach an acceptable compromise, for example, *bargaining(john)* that could support *reduced_sentence(john)* or *trial_transfer(john)* that could support *statute_of_limitation(john)*.

Thus, it would be nice leave to a machine reach the best goal for a given context. It would be necessary to define a new heuristic search exploration in a tree where the root is the initial keyclaim, and its child nodes are the best alternative to the keyclaim whose child nodes are the supporting arguments. In this case generate alternative keyclaim could be performed using a probabilistic approach or better, with a neural network able to meta-reason on the laws a choose the more suitable solution for the root keyclaim fail.

5.9 Conclusions

In this fundamental chapter we described all the implementation process of the thesis. We talked about Maggie MI and we provided part of the source code, Ubongo GUI its structure and its behavior. After testing this application we reach some demotivating results due to excessive resources usage, thus, we provided a deep analysis on the complexity of the system and we showed some alternative approach to reduce time and memory usage. Furthermore, for completeness's sake, we suggested some possible code optimizations regarding both parallelism and recursion optimization.

Chapter 6

Case study

In this chapter we present three case-studies, we prepared the first one on the basis of a typical example of meta-defeasible-reasoning known as "Villa example", the latter, instead, has been built on tweety the bird and acid cut approach. The former involves a complete remark of all the work we presented in the previous chapters and a deeper investigation on superiority_relation reasoning (that I leaved aside in the fifth chapter on purpose to explain using this simple example). The second one is important to see which are the condition that make able the tree to prune some of its branches in order to achieve a reasoning more similar to the human one and a way to reduce the complexity of the tree building process.

6.1 Villa example

Sartor and Prakken in [39] introduced their study regarding *defeasible priorities* and pronounced some criteria for their definition and how to reach inconsistent paradoxical state caused by their intrinsic weaknesses (as well as whatever else language definition).

In Chapter 5 we explained our *superiority relations* implementation, but what we said is not enough to grasp the meaning of a more complex case in which *superiority relations* can be multiple, nested or even paradoxical.

Actually, villa example represent the limit for an acceptable reasoning, and we will see one more example in which Maggie comes to a grinding halt since a paradoxical loop doesn't allow our meta-interpreter to reach the right conclusion.

First, we discuss villa example whose meaning is related to *Lex Prioriter Principium:*

Rule	Meaning
$r_1(x)$:	*x* is a protected building $\Rightarrow \neg x$'s exterior may be modified
$r_2(x)$:	*x* needs restructuring $\Rightarrow x$'s exterior may be modified
$r_3(y, x)$:	*x* is a rule about the protection of artistic buildings $\wedge$ *y* is a town planning rule $\Rightarrow y \prec x$
$T(x, y)$:	*x* is earlier than *y* $\Rightarrow x \prec y$
$r_4(r_1(x))$:	$\Rightarrow r_1(x)$ is a rule about protection of artistic buildings
$r_5(r_2(x))$:	$\Rightarrow r_2(x)$ is a town planning rule
$r_6(r_1(x), r_2(y))$:	$\Rightarrow r_1(x)$ is earlier than $r_2(y)$
$r_7(Villa)$:	*Villa* is a protected building
$r_8(Villa)$:	*Villa* needs restructuring
$r_9(T(x, y), r_3(x, y))$:	$T(x, y)$ is earlier than $r_3(x, y)$

Table 6.1: Villa example formalization.

> ***Lex Prioriter Principium:*** *if a law L_1 is posterior to another law L_2, then L_1 is superior to L_2.*

Thus, we will use this principle as superiority criterion between two rules for villa example definition. Now we will show how Sartor&Prakken formalized it:

> *"[...] They state contradicting priorities between a town planning rule saying that if a building needs restructuring, its exterior may be modified, and an earlier, and conflicting, artistic-buildings rule saying that if a building is on the list of protected buildings, its exterior may not be modified. [...]"[39]*

6.2 Theory definition

Furthermore, they provide a programming-like definition (Table 6.1)[1] that is the starting point for our prolog implementation. According with the syntax we mentioned in Chapter 5 now we show how to build a theory capable with Maggie, first we define a common KB:

[1]Where $y \prec x$ means x is superior to y.

```
neg renovate(X):= protected(X).
artistic_build((neg renovate(X):= protected(X))).
town_plan((renovate(X):= need_ren(X))).
protected(villa).
need_ren(villa).
%LEX PRIORITER PRINCIPIUM
sup(X,Y):= posterior(X,Y).
```

Later, we need to initialize the debate defining keyclaim and rules for opponent and proponent:

```
rule(key, (neg renovate(villa))).
rule(opp, (renovate(X):= need_ren(X))).
rule(pro, (sup(X,Y):= artistic_build(X), town_plan(Y))).
rule(opp, (posterior((renovate(X) := need_ren(X)),(neg renovate(X)
    := protected(X))))).
rule(pro, (posterior((sup(X,Y):= artistic_build(X), town_plan(Y))
   ,(sup(X,Y) := posterior(X,Y))))).
rule(mess, write('Villa example loaded')).
```

The keyclaim is "*neg renovate(villa)*", it means that tree derivation will be created only if the keyclaim is defeasible, and since we are putting forward the same number of rules from both the parties, according to Definition2 on page 88 , if in the first branch keyclaim will be derivable, it will be derivable for all the others branches. Running the example we get as a result a tree (Figure 6.1 tree's size was too big to enter in a single screenshot) that confirms what we expected.

To better understand what happened in our engine let's explain step by step how Maggie reacts to each assertion:

1. At the beginning asking for the keyclaim Maggie will answer: "*Yes*", instead for keyclaim negation will answer "*No*";

2. Later we assert "*(renovate(X):= need_ren(X)).*" and Maggie will answer Yes for both keyclaim and its negation[2];

3. When proponent asserts :

[2]Unless we are using a skeptical semantic.

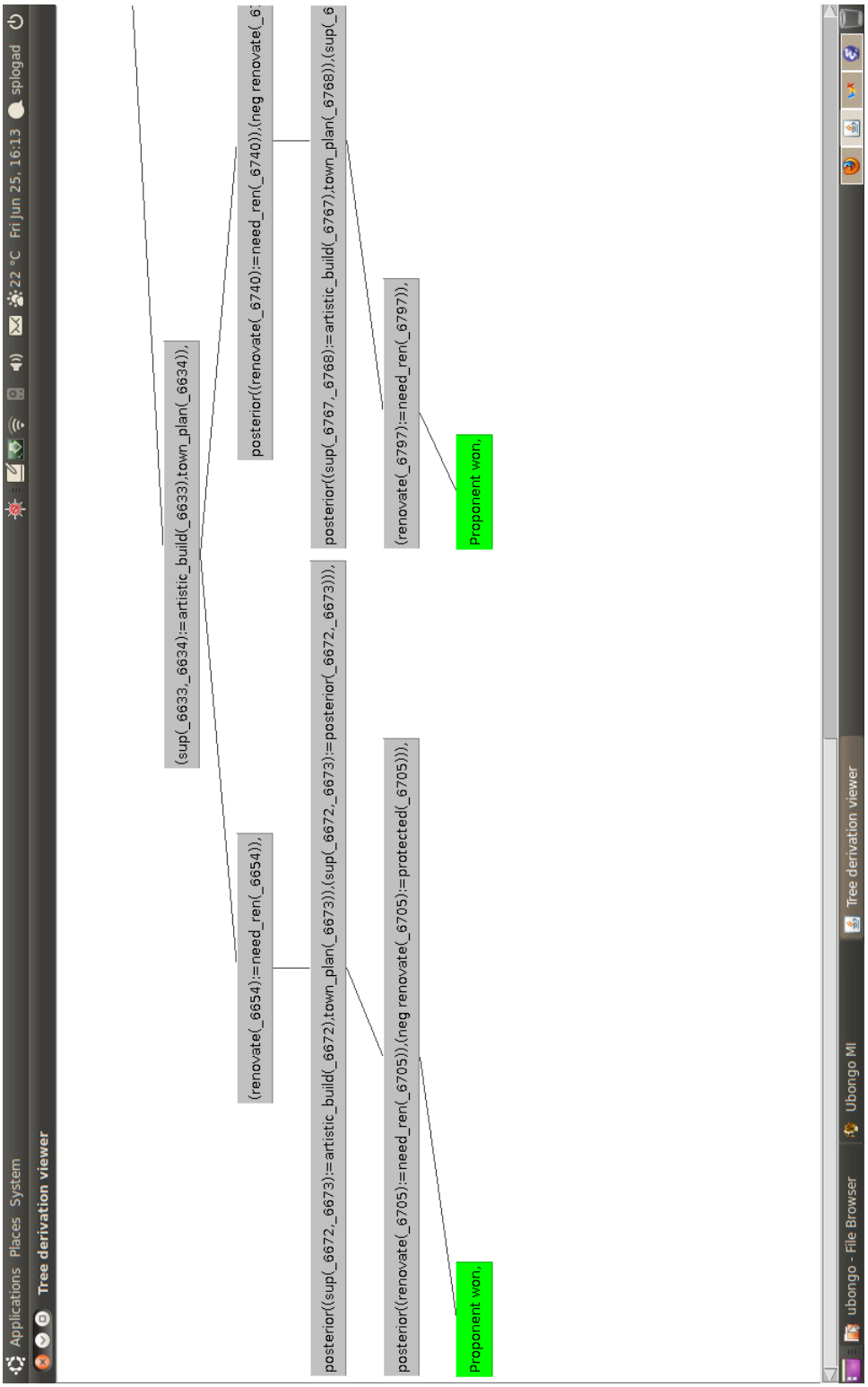


Figure 6.1: TreeViewer screenshot.

" *(sup(X,Y):= artistic_build(X), town_plan(Y)))."* since there is only a superiority relation and it is in favor of proponent, system will come back to say that only keyclaim is derivable;

4. Then, opponents puts forward " *(posterior((renovate(X) := need_ren(X)), (neg renovate(X) := protected(X)))))."* it means asserting a superiority relation since this rule matches with Lex Prioriter Principium's condition. The result will be the same of point 2, both keyclaim and its negation are derivable.

5. Now proponent asserts its *MatchPoint,* indeed, this rule is the only superiority relation between two superiority relations and it states that only keyclaim is derivable.

Last rule structure can look strange since asserting a *posterior/2* rule we indirectly state a superiority relation because the rule matches with `sup(X,Y) := posterior(X,Y).` The heart of the matter is that one of the arguments, in particular the second one, is the superiority relation itself, an we can write it in full:

```
sup((sup(X,Y):= artistic_build(X), town_plan(Y)),
(sup(X,Y) := posterior(X,Y)):=
posterior((sup(X,Y):= artistic_build(X), town_plan(Y)),
(sup(X,Y) := posterior(X,Y))).
```

But, remembering what stated Prakken&Sartor:

> "*Although this seems to be self-referential, formally, its in not, since it is one instance of Lex Posterior that speaks about another instance of itself*"[39]

it should be clear that it is not a paradox, indeed, during evaluation the variable X and Y will be bound by some value that make this rule just an instance and not a change of the superiority relation; in other words, in this case we don't need a meta-level evaluation since we are in the limit of a logic evaluation since, although, the rule is self-referred, it is a change of behavior in a specific case and it is not a rule of thumb.

Note that, since we putted forward rules with unbound variables, unification in prolog takes more time, instead in Tweety example all the rules introduced into the system were bound and performance was quite better (Table 6.2)

Example	Elapsed time
Tweety	0.040s
Villa	0.570s

Table 6.2: Bound and unbound variables performance test.

Going back to previous topic, we mean superiority relations and their self-reference we can show a case where Maggie falls in an infinite loop.

Before we show a limit-case of MI-self evaluation since the ambiguous rule didn't affect the system, but just a single instance, but, let suppose that we exploit Lex Prioriter to state its contrary.

1. Lex Prioriter Principium states that a posterior rule is superior to a earlier law;
2. I put a new rule into the system, say R, that is posterior to Lex Prioriter Principium;
3. R states that an earlier rule is superior to a posterior rule, and that is true because there exists Lex Prioriter that states the contrary.

This case is quite interesting since we are not dealing with specific instance, we are putting a new rule completely different that make us unable to reach a solution:

- We can reach a reasonable conclusion without using a reasonable reasoning;
- Or use a reasonable reasoning without reach a conclusion.

To better understand this kind of problem we need to survey a superiority relation's side effect. Indeed, in Maggie we defined all the rules in the same way, in particular, using two kinds of negation, but superiority relations require a new kind of negation. Let suppose we have a rule such as *sup(X,Y):= condition1(X), condition2(Y),* and leaving aside counter-attacks, we can assert the following kinds of negation:

1. `\+ sup(X,Y):= condition1(X), condition2(Y):` it is true if that rule fail;
2. `neg sup(X,Y):= condition1(X), condition2(Y):` it is true if there exists the explicit negation of the rule;
3. `sup(Y,X):= condition1(X), condition2(Y):` it isn't an explicit negation, we are just arguing that the contrary is true.

Intrinsic structure of superiority relation requires that a superiority criteria is valid only if it is proved that other superiority relation with different conclusion don't exist. But, in this case we reach an infinite loop since both superiority relations are valid and during evaluation the first one unify with a sup_rule that make it able to reach a conclusion, but trying to derive if the opposite fails it has to evaluate one more time itself as required by the opposite rule. In other words, asserting such as a rule the system enters in a mutual infinite recursion.

If we assert in villa example the following rules:

```
1 % Anti-Lex Prioriter
2 sup(Y,X) := posterior(X,Y).
3 posterior((sup(Y,X) := posterior(X,Y)),(sup(X,Y) := posterior(X,Y)))).
```

Maggie fails since is not able to reach a conclusion because of problems we explained above.

6.2.1 Solving antinomies

So far we said that the problem is a paradox, but it is not completely true, it's a particular kind of paradox called *antinomy*

> **antinomy**: *we get an antinomy when a logic reasoning produces in a correct way two solutions having the shape of thesis and antithesis, and they are both consistent and reliable, but they lead us to sentences such as : "A is true only if A is false".*

Our results are rather similar to antinomies, and we should understand which are the causes. If we analyze the last rules we proposed we notice that difference between Lex Prioriter and the rule we called *Anti-Lex Prioriter* is the position of the argument, instead, the condition remains the same. Maybe the reader could argue that is approach is a specific one, and that it depends on our syntax. Instead, we believe that this is the easiest way to reach this conclusion but it is not lack of abstraction. We could reach the same goal using different condition but it is only more complicated but it is the same.

Thus, taking into account that we are dealing with two rules with the same condition and opposite meaning, and remarking on point three of negation kinds list, we can argue that we are just dealing with a different kind of negation, but, in this case we are not using those rules to directly reach a conclusion, we are using them to reason on their priority relations.

It means, that in this form of antinomy we are not able to reach the final conclusion since each superiority check requires the execution of the other ones.

Roughly speaking, rules just added into the system, and in particular the first one, is simply conflicting with Lex Prioriter. Our rules are in the following form:

LexPrioriter: `sup(X,Y) := posterior(X,Y).`

Anti-LexPrioriter: `sup(Y,X) := posterior(X,Y).`

Quoting a physical approach we can define a new reference system in which *true* condition is replaced by *posterior(X,Y)* or more generally, by the current condition we are evaluating. We mean that those criteria will be considered only when their condition is true, and since they have the same condition, it doesn't make difference, namely, it cannot happen that only one of them is valid.

Now we can write one more time those two rules:

LexPrioriter$_0$: `sup(X,Y) :=` $true_0$`.`

Anti-LexPrioriter$_0$: `sup(Y,X) :=` $true_0$`.`

Remembering that inversion of arguments in a superiority relation means our new kind of negation, this problem can be considered as defining and asserting two opposite rules:

LexPrioriter$_0$: `<sup_rule> :=` $true_0$`.`

Anti-LexPrioriter$_0$: `neg`$_0$ `<sup_rule> :=` $true_0$`.`

But this is a known case, and this is the only rejecting criteria we defined, namely, we cannot assert a rule and its contrary. Obviously also if this intuitive proof seems to be applicable just in this case, we believe that it can be accepted as abstract:

Corollary 5. *Given a superiority relation R1*$\in CKB$ *of the form:* `sup(X,Y):=Condition.` *and let R2 be a new superiority relation to put forward into CKB whose condition is the same of R1, and its arguments are the same of R1 but with inverted position, R2 must be rejected.*

Thus, if Corollary 5 is not respected, our system could accept undecidability states. It means that in our meta-control of information consistency we have to add a new check that

applies Corollary 4. Notice that this approach is not only an elegant way to switch that issue from logic inconsistency to information inconsistency, but it is also useful since we prevent the evaluation at "*run-time*" rejecting the rule as soon as one of the player tries to put it forward. Rejecting a rule our tree will reduce the numbers of nodes to be evaluated, and it means a faster way to get a result as regards allowing the assertion of the inconsistent rule and sequentially evaluating if it is the case to use that rule.

6.3 Acid cut

In this second example we are showing the implementation of Acid cut as a form of complexity reduction in the tree build process. We propose to get the most interesting results merging acid cuts with different numbers of rules and or/or tree to see a nice improvement of the system.

First we define a new theory that represent the typical tweety's example but with some new features:

Common KB:

```
flies(X):=  bird(X).
neg flies(X):^ sick(X),bird(X).
bird(tweety).
bird(X):- penguin(X).
```

Private repositories:

Proponent	Opponent
strong(tweety). healed(tweety). useless(tweety). **flies(tweety).**	sick(tweety). brokenwing(tweety).

Table 6.3: Private repositories for tweety example with acid-cut and OR/OR derivation tree.

Furthermore, in Ubongo, we can try two different execution behaviors using the following predicate:

```
rule(or_and_tree, <on/off>).
```

Example	Elapsed time
Tweety (with OR/AND) 4X2 rules	0.020s
Tweety (with OR/OR) 4X2 rules	0.070s
Tweety (with OR/OR) 4X3 rules	0.310s

Table 6.4: OR/AND vs OR/OR performance test.

to choose which kind of derivation is the best for our purpose.

The theory we defined above is almost meaningless, but we need a theory that describe both cases of won an lost for Proponent. If we use proponent's last rule from table 6.3, namely, its match point, proponent wins, otherwise he loses. This is not possible to be represented in a OR / AND tree and using `rule(or_and_tree, on).` predicate (or without specification since OR/AND tree is the default) we get the result showed in picture 6.2 (a) representing a *Draw*[3].

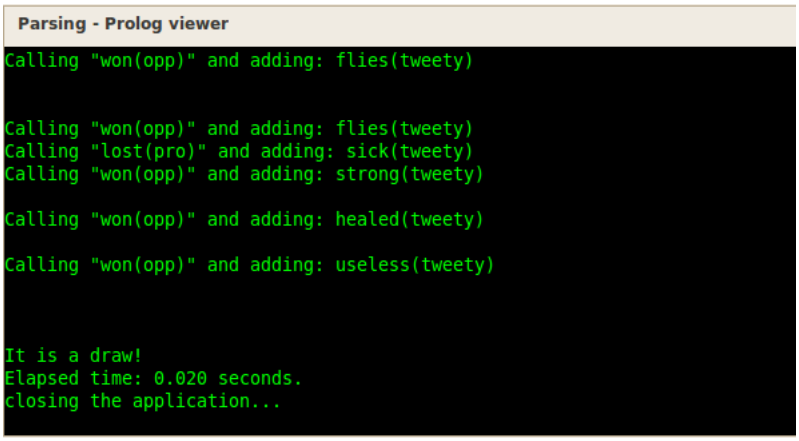
If we try to switch to OR/OR tree derivation, we notice that Ubongo draws a derivation tree that looks strange, since is pruned and doesn't contain all the branches (see figure 6.2 (b)). This representation is correct since branches pruning is due to acid cuts, and branches that are not showed into the draw are the branches that cannot return their value to the parent node since they failed (and so they won't be painted into the draw).

We know that the acid cut we putted is meaningless, but we just want to show that Maggie is able to deal with those kinds of cuts and that performance grows up. Analyzing the results showed in table 6.4, we notice that time spent for each operations is less than half second also if we are dealing with 4 rules vs 3 rules. Notice that first invocation took just 0.020s because as soon a branch fails, all the derivation tree fails, instead, in the second row (second operation) Maggie requires 0.070s but reaching a result and returning the list that Ubongo can use to draw the derivation tree.

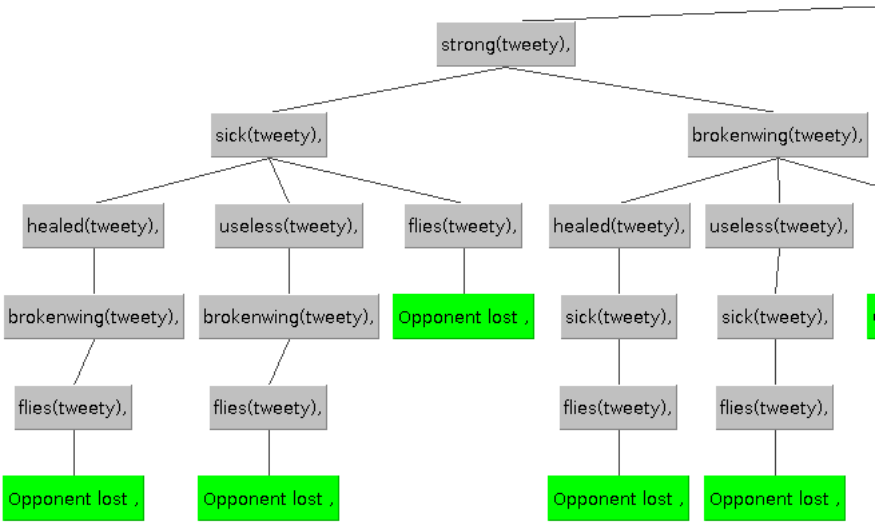
6.4 Reasoning on rules to add

In this last case study we show a remarkable improvement in derivation tree task in both complexity reduction and quality of reasoning. In Chapter 5 we presented a possible way to decide which rule is the most appropriate to be put forward in order to avoid an heavy

[3]Note that both tree derivations return draw since both trees contain cases of won and cases of lost.



(a) OR/AND tree derivation returns a Draw.



(b) OR/OR tree derivation shows branches where proponent won.

Figure 6.2: OR/AND vs OR/OR comparison.

resources usage and reach human behavior in term of making a decision in a given context rather than randomly try all the possible options.

In this section we briefly explain a new version of Maggie, called Experimental Maggie. This version provides a new kind of reasoning while the system has to decide if it is the case to assert a given rule. Furthermore, this version provide just proponent winning criteria tree with an OR/OR derivation tree.

We used a theory inspired on *Tweety Example* in order to make things easier to comprehend for the reader. This theory is quite similar to the one we saw before but there are some differences that we want to underline:

1. Players put forward rules with a structured Body;
2. Terms that only belongs to rules' conditions are considered just *supporting arguments* and are not put forward by Players;

We decided to change this behavior, since we believe that is better that a player be able to put forward an argument rather than a simple prolog rule, in other words, we hope that in a future release of Maggie we could integrate moves for both proponent and opponent containing a rule and its supporting arguments. Let the following example:

> *if a player wants to say that tweety can flies since it has got wings and it is a bird we would need the following rules:*
>
> - `flies(X) := has_wings(X), bird(X).`
> - `has_wings(tweety).`
> - `bird(tweety).`
>
> *It means that we need three moves to assert an argument. But, we could assert an argument with its supporting terms, in this case we would be able to assert something that is "self-explained" and doesn't require additional moves to be proved.*

As we said in Chapter 5, we would like to reason on the rules, in this case we deal with arguments but it isn't a problem since we can apply the same approach since reasoning on an argument would mean evaluate both *Head* and *Body* of the rule, and where *Body* contains *Heads* of supporting arguments. Anyway, this is just a suggestion for a future work, our aim is different. Let the following text be theory we want to evaluate:

```
CKB:
flies(X):=  bird(X).
bird(tweety).
sick(tweety).
healed(tweety).
brokenwing(tweety).
not-serious(tweety).
rule(key,flies(tweety)).
PKB:
rule(opp,(neg flies(X):^ sick(X),bird(X))).
rule(pro,(flies(X):= sick(X),bird(X),healed(X))).
rule(opp,(sup((neg flies(X):^ sick(X),bird(X)),
    (flies(X):= sick(X),bird(X),healed(X))):= brokenwing(X))).
rule(pro,(neg sup((neg flies(X):^ sick(X),bird(X)),
    (flies(X):= sick(X),bird(X),healed(X))):=
       brokenwing(X), not_serious(X))).
```

All supporting arguments belongs to CKB just because we don't want to make the system busy evaluating them in different nodes. We used Recommended rules criteria showed in Section 5.6.1, namely, given a Rule that enemy put forward in previous step, Proponent has to defend its keyclaim, Opponent has to attack keyclaim.

We propose a simple algorithm that, given previous rule, say *LastRule,* and a given rule *R* from private repository, it states that *R* is *Recommended* if:

1. *R*'s conclusion is the contrary of *LastRule's* conclusion.
2. *R*'s conclusion is the contrary of a Term in *LastRule's* condition able to change *LastRule's* conclusion derivability.
3. Be *R* a superiority relation; if asserting *R* Keyclaim derivability is proved (for proponent) / is not proved(for opponent), then *R* can be added.

It's quite easy understand the meaning of this algorithm since we are trying to put forward the most suitable rule in a given context.

Running Maggie-Experimental we reach some interesting results. Indeed, while system is trying to add a rule that be appropriate for the context rather than deriving randomly all

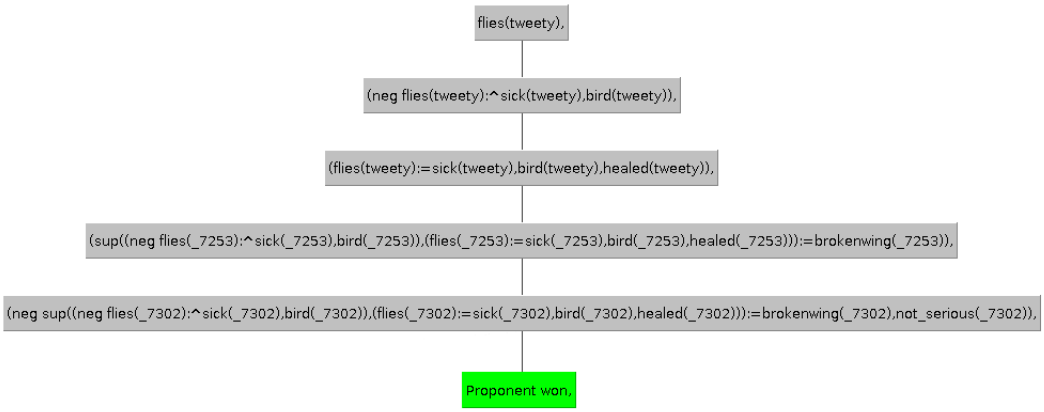


Figure 6.3: Reduced tree size using *Recommended rules.*

the results, size tree is quite reduced and now is just a single branch. It's a nice result since not only we reduced complexity, but also because reading the tree its derivation is really similar to human reasoning.

In Figure 6.3 we can see the screen-shot of Maggie-Experimental execution. Since Proponent proposed the *keyclaim*: "*flies(tweety)*", the most appropriate rule for opponent is a rule more specific that assert the negation of keyclaim. Indeed, Opponent added a more specific defeater. Later, Proponent used the same reasoning on the Opponent attack and puts forward a rule more specific of the attack that support keyclaim.

After those two rules, players can only use superiority relation and since are both valid in the steps in which they are evaluated, both will be added. Proponent won since managed to defend keyclaim and proponent exhausted rules to assert.

6.5 Conclusions

In this section we presented two examples that represent really important cases of study. In the second examples we proved that we can cope superiority relation inconsistency preventing the evaluation and switching the problem form logic inconsistency to information inconsistency. This approach represents an important step for the evaluation task of our system. Indeed, it means that we can avoid not only the meta-evaluation, but also reduce the number of nodes in the derivation tree since some rules are not consistent with CKB. Furthermore, we sheds light on a further approach able to reduce complexity of our tree, we

mean, *Acid Cut.* We remark that due to lack of time we cannot provide a better definition for *Acid Cutting* criteria, and so we can only suggest to researches a new interesting way to make the application faster than before.

Chapter 7

Conclusions

In this chapter we will discuss the work presented in this thesis. We will first give a brief summary of Maggie Meta-Interpreter and Game-model implementation. We will then discuss which improvements on argumentative debate and MIs have been made and have a closer look at parts that could have been done in a different and maybe better way. Later we will show some suggestions for future works in order to improve the work we presented in the previous chapters and make it able to cope a "real" debate with better performance and bring this system more near to human reasoning.

7.1 Summary

In this work we dealt with knowledge representations, non-monotonic logics and a game-model that make possible building a human reasoning simulator able to contend a legal debate between two parties. In the first part of the thesis we presented background materials, we showed the state of the art of non-ground MI and we present a possible game-model. In particular, we focused on the lack of implementations that complicate theoretical approaches testing.

We proposed a Meta-Interpreter, called Maggie, that provides a lot of features proposed over the years such as defeasible superiority relations and we extended the notion of defeasible derivability on rules that before were just considered strict. Our aims in this thesis is not only to show MI implementation, but also use it as under-layered framework to perform an argumentative debate based on the game-model defined in Chapter 5.

The study of such topics can be useful to analyze human reasoning and its applications in different domains such as *psychology*, *philosophy* and *legal debate*, furthermore, since tree derivation model intrinsically involves a parallel search of a conclusion in different contexts (where a context represents a *random* assertion of rules during the evaluation steps in a different order), our work also provides an helpful model for *financial* in term of strategies investigation.

We defined our meta-interpreter starting from a set of rules and a given keyclaim that is the goal to successfully reach using the set of rules, and defeasible logic-based predicates (that we show on page 133).Set of rules we just defined is called *theory* in we can distinguish in three subsets: *CKB* that is the common knowledge and both parties are aware of it, *ProKB* is a private subset of rules that is known only by the *proponent* of the game model and *OppKB* is the private subset of rules that belongs to *Proponent* of the game model. As you can see the assertion of those rules is not a duty of MI, but we need a new meta-level able to manipulate those rules and use Maggie-MI just to evaluate the keyclaim. Thus, game-model meta-level performs the challenge between the players and manage their private repositories as we showed in Chapter 5. We need to remind that this application has been conceived to be modular in the sense that there are not strict dependencies among meta-levels, we mean that on the one hand MI can be used with a different game-model whose burden is just to know the name of the predicates of the meta-interpreter, on the other hand we can use only MI (without game-mode) through a Prolog editor such as Emacs and execute goals we prefer like in whatever else Prolog software.

In this work we tried to demonstrate that the application of a game-model, such as the one described in [38], could be dramatically complex and leads the system to an excessive usage of memory and processor that jeopardize time performance. A deeper evaluation of the game-model complexity will be summarized in the next section.

7.2 Evaluation

In this section we evaluate results of our work and compare them with objectives that we had set ourselves and check if they have been achieved. As we said in Chapters 5 and 6 we got some problems, in term of performance, running the game model since it involves all the rules we defined in a tree-structure that is built recursively. Unfortunately, recursion can be really slow since it requires that, during a nested evaluation, all data-structure that

belongs to invoking method have to remain on the stack until the nested call has returned a result. It means that depending on the number of rules we are dealing with, tree complexity can grows much larger and it become impossible to reach a conclusion in an *acceptable* amount of time.

We propose some suggestions in order to reduce game-model complexity and a deep theoretical analysis that shed lights on intrinsically complexity since we will rich some all the possible conclusions both in the case in which it is not needed to build the whole tree as stated by Theorem 3, and in the case in which it would necessary survey all the nodes but the system could not be able since there is the limitation of and/or tree structure as shown in Theorem 2. We solved the latter problem since Maggie is now able to modify the kind of tree derivation, but to solve the former problem it would be better to rebuild the system in an iterative way in order to avoid the construction of the whole tree when is the case to use depth-first search.

About Maggie-MI, we noticed that its performance are quite satisfactory and it is able to solve all the known problem proposed in literature that we showed in Chapter 5 such as Tweety the Bird, Nixon Diamond and Villa Example. The latter one has been an important starting point for our work since during its analysis we proposed a new approach to solve paradoxes as we showed in Corollary 4 in the Chapter 6. We proposed to switch from logic inconsistency to information inconsistency check. In this way we can reduce computational load since we reject a rule that is inconsistent with CKB and so, we don't need to evaluate them during the normal derivability process of the keyclaim. This approach could be interesting also for loop detector, indeed, in Maggie this approach prevent a loop caused by a double assertion of a rule and its negation. In the next section we also show that this is not a particular case, then we could also apply the same approach to solve other kind of paradox such as *liar paradox, etc.*.

Finally, we provided an additional approach in order to reduce complexity that we called *Acid Cuts.* This approach could be interesting also for implementing in the future a reasoning way more similar to the human way and we discussed in Chapter 5. In the next section deserve a mention the reasoning criterion for putting forward a new rules, theoretically explained in [38] that, because of lack of time, we didn't implemented. Putting forwards rules without criteria is one of the cause of our *crazy* growth of complexity. In future work section we suggest some approach to prevent the creation of useless branches of the tree that could lead the system to noticeably reduce its complexity.

7.3 Future Works

In this section we gather up all the the suggestions for future work that we conceived during the writing thesis period. We can distinguish between the ones we proposed in order to reduce complexity and the others that proposed to improve our system and to have it behave more similar to humans beings.

We tried to *invent* some new approaches for the tree exploration in order to prune the tree and avoid that its size become huge also dealing with a limited set of rules in private repositories. It would be a nice addition to our system the implementation of a branches reduction technique as *Acid Cuts,* that we deeply explained in Chapter 5, but deserve a mention in this chapter since it can be useful in both the domains since we propose to use *Acid Cuts* to reduce complexity and get a result in a quicker way, but we also use this criterion to generate a mechanism more similar to human mind, namely, an approach to define a double mind approach that not only is able to accept some information as true defeasibly but can also transform those *defeasible* truths into *strict* truths. In this way we can reduce complexity and obtain an *approximate* result, and if, say with a feedback, we notice that given result is caused by a *wild*[1] *Acid Cuts*. In this case we can execute more times the execution of the debate with a growing complexity and hope for a result as soon as possible, namely, before that tree reach its whole size.

One more future work in order to solve complexity issues would be analyze and formalize a reasoning technique to choose the rule to put forward. As explained in [38], we should not allow *random* assertion of rules since it would be a demanding evaluation of all the possible permutations of the items of two sets of rules. So we should take from the repositories only the rules that can be used for a specific goal, namely, to demonstrate that keyclaim is defeasible (for the proponent) and to demonstrate that the negation of keyclaim is derivable or that keyclaim is not defeasible for the opponent.

Although it may seem an easy task to formalize that behavior, it would be really difficult since our system should be able to recognize the semantic meaning of a sentence, furthermore, depending on the selected ambiguity behavior, opponent's goal could change, since in ambiguity propagation domain proving that negation of keyclaim is derivable is not enough for winning the challenge. So we suggest to use, as starting points, criteria proposed in Chapter 5 in a future work and check if complexity reduction can be reduced as expected.

[1] With *wild Acid Cut* we mean a cut that it was not appropriate to do.

Unfortunately, lack of formalisms cannot help us to find a *deterministic* improvement since it will depends on both criteria we are using and theory to evaluate.

Leaving aside programming optimizations we would spent some words to remark *Multiple Keyclaim Approach* that we presented on Chapter 5. Using Multiple keyclaims we can cope well-constructed debate in which opponent/proponent's goals can be more flexible and change on the fly, say, like we expressed in the example on Chapter 5 to find a compromise by heuristics search and achieve a acceptable results although is different from the original keyclaim.

7.4 Conclusion

In this chapter we summarized the main topics of our work in order to evaluate and check if targets we had set ourselves at the beginning of the thesis have been achieved. We are quite satisfactory since we not only implemented a framework for defeasible logic and a game-model argumentative debate, but we also analyzed the limitation caused by performance issues and we proposed also some suggestions to solve those problems and ones more to improve the system.

List of Figures

List of Tables

List of Algorithms

Appendix A

Additional Material

A.1 Göedel theorem

When we quote Göedel theorem, we mean the second[1] of its theorem of completeness that state:

> *For any formal effectively generated theory T including basic arithmetical truths and also certain truths about formal provability, T includes a statement of its own consistency if and only if T is inconsistent.*

Without losing correctness and for simplicity's sake we can replace *formal effectively generated theory T* with *language L* and *including basic arithmetical truths* with *including basic logic truths.* In this way we can state that every language is not able to pronounce its features using itself. That is the reason why we need a meta-interpreter to avoid *paradoxes.*

Creating more meta-levels is like building some steps over our language that we have to reach in order to evaluate the lower level. If we are not dealing just with meta-interpreter, but some meta-levels, we could represent this structure of meta-evaluators not using a single staircase, but a staircase in which for each following step we can have more than one option depending on the kind of evaluation. It would be something rather similar to Escher staircase.

In both case we need a component that is located above our object-language and it is able to invoke *below* predicates. Maggie-MI will be a meta-interpreter since it is independent and

[1] Also if the second theorem of completeness is proved formalizing part of first theorem's demonstration.

(a) Meta-languages, going up a step we find a new independent meta-language able to evaluate the levels below

(b) Meta-levels, going up a step we find a two possible ways, we mean to kinds of evaluation depending on which is our purpose, for example just evaluate superiority relations.

Figure A.1: Different structure depending on we are dealing with meta-languages or meta-levels.

it isn't just a tool to use, it's able to reach a conclusion using *defeasible prolog language.* Game-model, superiority relations evaluator are just meta-level, since we use them (we invoke their predicates on superior level) just to reach a conclusion.

A.2 Meta-logic formalization

Now I define a meaningful and legal acceptable jurisprudence in a certain level, say T1 through a meta-level, say T2, of the meaningful and legal acceptable rules as instances or rules expressed using the meta-level theory language.

The relation between the theory and meta-theory, (T1 and T2) is defined as upward reflection rules:

$T1 \vdash A$

$(T2 \vdash (T1 \vdash A))$.

In our case:

$(T1 \vdash$ Receiving Messages $\Rightarrow$ privacy violation $\supset$

$T1 \vdash (KB\_Code \neg \vdash$ Receiving Messages $\Rightarrow$ privacy violation) &

$T1 \vdash (KB\_x \vdash$ Receiving Messages $\Rightarrow$ privacy violation) &

T1 $\vdash$ (KB_y $\vdash \neg$ Receiving Messages $\Rightarrow$ privacy violation)

where x,y are the two parties of the debate. In particular x is the user and y is Microsoft.

Using the Meta-theory I cannot reach a final state or a unique solution. That is the reason why we use the approach of a game between the parties Thus, we can obtain all the possible sequential states depending by the sentences (acts) uttered by the parties.

A.3 Negation and quantifier in Prolog

In this brief appendix we want to explain how the negation works in prolog. Actually can be strange, but sometimes using prolog we obtain different results using the same predicates in a different positions. We know that prolog (for performance reason) decided to adopt the depth-first research strategy, but it also important understand the non-*safe* selecting literal behavior during the negation task.

Prolog use the leftmost strategy while inspect the literals, so:

```
We say:
:- not(p(X)).
and its meaning is ∃ X(not p(X))
prolog check
:- p(X)
whose meaning is ∃ X(p(X))
So denies the result:
not(∃ X(p(X)))
whose meaning is:
∀ X(not p(X))
```

So if we use the following program:

```
unemployed(X) :- not employed(X), adult(X).                    (1)
employed(john).
adult(mary).
```

And the following query:

```
?- unemployed(X).
```

We want to know if exist X such that is unemployed. So we expect `yes, X/mario`, but the programs answer `no`.

Furthermore, using the query:

```
?- unemployed(mario).
```

It answers `yes`.

If we change (1) in this way:

```
"unemployed(X) :- adult(X), not employed(X)."
```

the program starts to works.

That's because in (1) it applies a non ground term, as "`not employed(X)`" is non *ground* during the evaluation.

A.4 Vanilla Meta-Interpreter

In this appendix we show a simple use of a Vanilla MI.Let this example expressing natural numbers through successor mechanism:

```
natnum1(0).
natnum1((s(X)):-natnum1(X).
```

using *clause* predicate, we can inspect the program:

```
?- clause(natnum(1),Body).
Z=0
Body = true;
Z= S(_G254));
Body = natnum(_G254);
No
```

A more complex sample is the following

```
complex_goal(A):- g1(A),g2(A),g3(A).
```

This clause has a body composed by more goals. They, in turn, are composed by goals or complex terms.

The following is a body example:

```
body(true).
body((A,B)):- body(A), body(B).
body(G):- goal(G).[2]
body(_=_).
body(call(_)).
...
```

The code above represent our program. Let's define an interpreter for this program:

```
m1(true).
m1((A,B)):-mi1(A, mi1(B)).
mi1(Goal):-
    Goal \=term,
    Goal \= (_,_),
    clause(Goal,Body),
    m1(Body).
```

This kind of interpreter is called ***Vanilla*** because it doesn't add particular features to our programs, but it can be helpful, because it's the backbone that we can use to improve our programs.

```
?- mi1(natnum(X)).
X=0;
X=s(0);
X=s(s(0))
Yes
```

The above example sheds light that *we have defined what is not a goal, instead defining what is a goal.*[3]

[2] Be careful, this clause matches with both *true* and *(_,_)*, it's really ambiguous.

[3] Such representation are called "*defaulty*".

Appendix B

Maggie code

B.1 Attacking predicates

B.1.1 Rebutted

```
1
2  rebutted(KB,(Head := Body)) :-
3          contrary(Head,Contrary),
4                  def_rule(KB,(Contrary := Condition)),
5                  def_der(KB,Condition),
6                  sup_rule((Contrary:= Condition),(Head:= Body)),
7                  !.
8
9  rebutted(KB,(Head := Body)) :-  ab,
10                 contrary(Head,Contrary),
11                 def_rule(KB,(Contrary := Condition)),
12                 def_der(KB,Condition),
13                 \+ sup_rule((Head:= Body),(Contrary:= Condition)).
```

B.1.2 Undercutted

```
1  undercut(KB,(Head := Body)) :-
2                  contrary(Head,Contrary),
3                  (Contrary :^ Condition),
4                  def_der(KB,Condition),
5                  \+ sup_rule((Head := Body),(Contrary :^ Condition)),
```

```
6                 !.
```

B.2 Priority_relation and specification

```
1  sup_rule(Rule1,Rule2) :-
2                 sup(Rule1,Rule2),!.
3
4  sup_rule(Rule1,Rule2) :- dp,
5                 clause(sup(Rule1,Rule2),Condition),
6                 Condition \= true,
7                 def_der(KB,sup(Rule1,Rule2)),
8                 !.
9
10 sup_rule(Rule1,Rule2) :- dp,
11                def_rule(KB,(sup(Rule1,Rule2) := Condition)),
12                def_der(KB,(sup(Rule1,Rule2))),
13                def_rule(KB,(sup(Rule2,Rule1) := Condition2)),
14                \+ def_der(KB,(sup(Rule2,Rule1))),
15                !.
16
17 sup_rule(Rule1,Rule2) :- dp,
18                def_rule(KB,(sup(Rule1,Rule2) := Condition)),
19                def_der(KB,sup(Rule1,Rule2)),
20                def_rule(KB,(sup(Rule2,Rule1) := Condition2)),
21                def_der(KB,sup(Rule2,Rule1)),
22                sup_rule((sup(Rule1,Rule2):= Condition),
23                (sup(Rule2,Rule1) := Condition2)),
24                def_der(KB, Condition2),
25                !.
26 %%% SPECIFICITY
27 sup_rule((_ := Body1),(_ := Body2)) :-
28                def_der(Body1,Body2),
29                \+ def_der(Body2,Body1),nl,
30                write(Body1),write('␣is␣superior␣to␣'),write(Body2).
31
32 sup_rule((_ := Body1),(_ :^ Body2)) :-
33                def_der(Body1,Body2),
34                \+ def_der(Body2,Body1),nl,
35                write(Body1),write('␣is␣superior␣to␣'),write(Body2).
```

B.3 Contrary and Incompatibility

```
1  contrary(Clause1,Clause2) :-
2                  incompatible(Clause1,Clause2).
3
4  contrary(Clause1,Clause2) :-
5                  incompatible(Clause2,Clause1).
6
7  contrary(Clause1, Clause2) :- dp,
8                  clause(incompatible(Clause1,Clause2),Condition),
9                  Condition \== true, !,
10                 def_der(KB, Condition).
11
12 contrary(Clause1, Clause2) :- dp,
13                 def_rule(KB,(incompatible(Clause1,Clause2) := Condition)
                       ),                        def_der(KB,incompatible(
                       Clause1,Clause2)),           def_der(KB,Condition).
14
15 contrary(Clause1, Clause2) :- dp,
16                 def_rule(KB,(incompatible(Clause2,Clause1) := Condition)
                       ),                        def_der(KB,incompatible(
                       Clause2,Clause1)),           def_der(KB,Condition).
17
18 contrary(Clause1,Clause2) :-
19                 atomicNeg(Clause1,Clause2).
```

B.4 Game-model won(proponent)

```
1  % Proponent won criteria.
2
3  won(pro,CKB, ProKB, OppKB,Tree,NewTree):-
4          %proponent wins if the keyclaim is strictly derivable.
5          keyclaim(Keyclaim),
6          assertall(CKB),
7          strict_der(KB, Keyclaim),
8          nl,write('Proponent won...strict!'),nl,
9          append(Tree,[['Proponent won',#,[]]],NewTree),
10         unassertall(CKB),!.
11 won(pro,CKB, ProKB, OppKB, Tree, NewTree):-
```

```
12          %it also wins if the opponents have not more moves
13          isEmpty(OppKB),
14          keyclaim(Keyclaim),
15          assertall(CKB),
16          def_der(KB, Keyclaim),
17          append(Tree,[['Proponent won',#,[]]],NewTree),
18          unassertall(CKB),!,
19          nl,write('keyclaim defeasible and opponent finished its rules!').
20
21  won(pro,CKB,_,_,_,_) :- unassertall(CKB),fail.
22
23  won(pro,CKB,ProKB,OppKB,Tree,NewTree):-
24          orand,
25          countPlayer(ProKB,ProItem),
26          findall([X,'#',NewTreeLost],(
27             member(X,ProKB),
28          move(X,CKB),
29             append(CKB,[X],NewCKB),
30             delete(X, ProKB, NewProKB),
31             nl,write('Calling "lost(opp)" and adding: '), write(X),
32             lost(opp,NewCKB,NewProKB,OppKB,Tree, NewTreeLost)),List),
33             append(Tree,List,NewTree),nl,
34          countPlayer(List,N),
35          N == ProItem.
36
37          won(pro,CKB,ProKB,OppKB,Tree,NewTree):-
38          \+      orand,
39          countPlayer(ProKB,ProItem),
40           findall([X,'#',NewTreeLost],(
41                  member(X,ProKB),
42              move(X,CKB),
43                  append(CKB,[X],NewCKB),
44              delete(X, ProKB, NewProKB),
45                  nl,write('Calling "lost(opp)" and adding: '), write(X),
46                  lost(opp,NewCKB,NewProKB,OppKB,Tree, NewTreeLost)),List),
47          append(Tree,List,NewTree),
48          countPlayer(List,N),
49          List \== [].
```

Index

Bibliography

[1] Keith Frankish. Non-monotonic inference. *Keith Brown, Hg.: The Encyclopedia of Language and Linguistics, Oxford:Elsevier*, 2005.

[2] P. Mello L. Console, E. Lamma. *La Programmazione Logica*. UTET, 2 edition, 1997.

[3] Maarten; Blackburn, Patrick; de Rijke and Yde Venema. *Modal logic*. Cambridge University Press, Cambridge, 1 edition, August 2001.

[4] Robert A. Kowalski. Problems and promises of computational logic. *Proc. Symp. on Comp. Logic, Springer*, pages 1–36, 1990.

[5] Susan G. Josephson John R. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, Cambridge, UK, 1995.

[6] T. Sebeok. *The Play of Musement (Advances in Semiotics)*, volume 384. Indiana Univ Pr, IA, 1st edition. (january 1982) edition, 1981.

[7] P. Codgnet. Abductive reasoing: Backward and forward. *IJCAI'97*, page 4, 1997.

[8] D. Poole. A logical framework for default reasoning. *Artificial Interlligence*, 36:27–47, 1988.

[9] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[10] R. Reiter. On asking what a database knows. *Proc. Symposium on Computational Logic, Springer-Verlag, Berlin*, pages 93–113, 1990.

[11] D. McDermott S. Hanks. Default reasoning, non-monotonic logics, and the frame problem. *Proc. AAAI,*, (Morgan and Kaufman):328–333, 1986.

[12] D. McDermott S. Hanks. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412., 1987.

[13] F. Sandri R.A. Kowalski. Logic programs with exceptions. *Proceedings of the Seventh International Conference on Logic Programming*, (MIT Press, Boston):598–613, 1990.

[14] Peter Abelard. *Sic et Non: A critical Edition*. University of Chicago Press, Chicago, 1976.

[15] Lynn E. Rose. *Aristotle's Syllogistic.* Charles C Thomas Publisher., Springfield, 1968.

[16] Edith & Cairns Hamilton. *The Collected Dialogues of Plato, Including the Letters*. Princeton Univ. Press, Princeton, huntington (eds.) edition, 1961.

[17] R Loui C. Chesñevar, A Maguitman. Logical model of argument. *ACM Comput.Surv.*, 32, 4:337–383, 2000.

[18] S. Toulmin. *The Uses of Arguments*. Cambridge Univ.nPress, Cambridge, MA, 1958.

[19] J. Pollok. *Knowledge and Justification*. Princeton University Press, Princeton, N.J., 1974.

[20] P. M. Dung. On the acceptability of arguments and its foundamental role in nonmonotonic reasoning and logic programming. *Proc. of the 13th International Joint Conference in Artificial Interllignence*, 1993.

[21] S. Alvarado. Understanding editorial text: A computer model of argument comprehension. *Kluwer Academic Publishers.*, 1990.

[22] Rescher. Dialectics: A controversy oriented approach to the theory of knowledge. 1977.

[23] Morgan Kaufmann Publishers. Argument systems: a uniform basis for nonmonotonic reasoning. *Proceedings of the 1st International Conferenceon Knowledge Representation and Reasoning*, pages 245–255, 1989.

[24] R.P. Loui G.R. Simari. A mathemati-cal treatment of defeasible reasoning and its implementation. *Artificial Intelligence 53*, pages 125–157, 1992.

[25] P.M. Dung. An argumentation semantics for logic programming with explicit negation. *Proceedings ICLP'93*, (MIT Press):616–630, 1993.

[26] P.M. Dung. On the acceptability of arguments and its fundamental role in nomonotonic reasoning and logic programming. *Pro- ceedings of the 13th International Joint Conference in Artificial Intelligence (IJCAI)*, (Chambery,France):852–857, 1993.

[27] G. Sartor H. Prakken. A system for defeasible argumentation, with defeasible priorities. *Proceedings of the International Conference on Formal Aspects of Practical Reasoning, Springer Verlag*, (Bonn, Germany), 1996.

[28] M. Giacomin P. Baroni. Semantics of abstract argument systems. *Argumentation of Artifical Intelligence*, 2009.

[29] R. Kowalski F. Toni A. Bondarenko, P. Dung. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence 93*, pages 63–101, 1997.

[30] D. Poole. A logical framework for default reasoning. *Artificial Intelligence 36*, 1:27–47, 1988.

[31] P. M. Dung. An argumentation semantics for logic programming with explicit negation. *Proceedings ICLP'93*, (MIT Press):616–630, 1993.

[32] J. Fischer Nilsson A. Hamfelt J. Eriksson Lundström. Legal rules and argumentation in a metalogic framework. *Frontiers in Artificial Intelligence and Applications*, 2007.

[33] J.W. Lloyd P.M. Hill. Analysis of meta-programs. *Meta programming in logic programming, MIT Press Cambridge, MA, USA*, pages 23–51, 1989.

[34] J.W. Lloyd P.M. Hill. Analysis of meta-programs. *Meta programming in logic programming*, 1989.

[35] Jan Maluszynski Ulf Nilsson. *Logic, Programming and Prolog*. John Wiley & Sons Ltd, 2nd edition, 2004.

[36] R. Bartak. *Prolog Programming*. Charles University, Prague, 2 edition, 1998.

[37] A. Vellino M.A. Covington, D. Nute. *Prolog Programming in Depth*. September 1995.

[38] J.S.Z. Eriksson Lundstrom. *On the Formal Modeling of Games of Language and Adversarial Argumentation. A Logic-Based Artificial Intelligence Approach.* Universitetstryckeriet, Uppsala, 2009.

[39] G. Sartor H. Prakken. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7:25–75, 1999.

Ringraziamenti

In questa sezioni in maniera a dir poco non-formale cercherò di ringraziare tutte le persone che hanno contribuito nel folle tentativo di supportarmi durante questi anni di faticosi studi.

Tanto per cominciare un gran enorme grazie ai miei genitori; senza di loro oggi mi ritroverei come un eremita su qualche albero, anzi senza di loro probabilmente non mi ritroverei affatto. Un grazie anche a mia sorella Stefania e al caro vecchio Vincenzo che mi hanno aiutato ed ospitato in quel di Rimini varie volte nonchè imposto a poveri piloti d'aereo di salturmi in fase di atterraggio grazie :). Un mega-grazie a Virginia che mi ha dato un mano a diventare una persona più consistente, mi ha comprato i biglietti per la vacanza e mi ha dato casa, in pratica fa tutto lei. Un mega grazie a tutte le persone che ho incontrato a Bologna (ma anche a quelle che ho conosciuto prima) durante questi *[1] anni e con le quali ho affrontato una tappa edificante e spesso alcolica della mia "*tarda adolescenza*" tanto per cominciare la cara Antonella con *annessa* sorellina Giovanna[2] amica di tante (dis)avventure ArezzoWave e soprattutto infinite!! chiacchiere e molto altro ancora. Come dimenticare i goffi approcci in cerca di amici dei primi giorni Bolognesi? Proprio allora mi è capitato di avere a che fare nell'ordine con il buon Cascio (non rivelerò l'esatto contesto d'esame in cui ci siamo conosciuti...) e l'eternamente strano Danilo, non temere: un giorno torneremo a cantare *La canzone dell'amore perduto* sotto i portici con la bicicletta e con la spenzieratezza della gioventù. Quanti bei ricordi: partitelle in piazza san francesco prima che la specialistica ci succhiasse via la vita dal midollo (cit. *o quasi*). Grazie alla comitivella di quei bei tempi con Davide, Federica, Alessia y la compañera Alexandra. A dire il vero temporalmente mi preme ringraziare i primi veri amici d'università che erano anche amici di liceo, come dimenticare la prima casa di Ferrara in doppia col povero Giacinto che

[1] Non lo dico per vergogna...

[2] Lo so è dura sentirsi definire fratello o sorella di qualcunaltro, ma è lo scotto che si paga ad essere fratelli minori...tuttosommato siamo più intelligenti, quindi, è un compromesso accettabile no?

mi ha dovuto sopportare nel mio esordio nel magico mondo della follia, il povero Rigoni sempre in macchina direzione Piemonte, anzi no Padova, perchè non Termoli? e l'etnico Pandoro per cena dove lo mettiamo!??!? Biondaaa!!! il caro vecchio Gvn che non sento da un casino, amico storico nonchè il miglior batterista di tutti i tempi...bhè dopotutto ha suonato nei gloriosi Sapor Vinga[3]. Giacchè ci sono ringrazio i compagni di musica dei sapor vingla: Francesco, auguri per i tuoi album...a te che hai avuto il coraggio di seguire l'ardua strada del musicista nonostante una laurea in ingegneria ambientale; Giovanni (l'unico che si becca ringraziamento doppio...ti è andata bene) e l'ultimo arrivato ma non meno importante (benchè a stento oggi mi saluti) Giacomo[4]. Grazie ai Termolesi con i quali ho praticamente i rapporti compromessi :), pole position per Valerio, amico eterno e dalle mille risorse nonchè iniziatore della mia *carriere* musicale, un grazie anche a Valeria che mi ha portato in giro per Bologna i primi mesi della mia permanenza, il caro buon Rosati...dove sarai ora!?!? San Diego giusto? hehhe un grazie a distanza! a Daniela grande appassionata dei Sapor Vingla, lo so erano irresistibili. Ancora un grazie a Antonio, si il pianista iuliano, Flavia, Marianna tutta pozzo dolce e la cassa quindi Luigi, Barone, Gino, Tania e i mille caffè domenicali e Chiara (i tuoi ultimi lavori sono superiori)[5].

Meriterebbe una sezione a parte il ringraziamento all'esperienza di Madrid un saluto alla *collega* Miriam, un giorno riusciremo a rivoluzionare il mondo tecnologico con le nostre folli idee informatiche, Valentina e il locale delle borsette col brasiliano logorroico nonchè la parete da ripitturare, il buon Federico che mi ha raggiunto fino a Stoccolma, i miei conquilini Adela, Julia e Tiago. Grazie anche ai compagni di spagnolo in primis Lou e Laure che figata la barca a stoccolma, intendo l'ostello NON il vasa! un particolare ringraziamento alla sangria della Viga e a don Alejandro.

Grazie anche agli ultimi coinquilini di Bologna Giuliana, Michela, Chiara (non sarai mica ancora a spasso per la penisola balcanica?!?!) e Ernesto (ogni tanto ripenso ancora al film di Bressan :)).

Ultimi, ma solo cronologicamente parlando, gli ultimi personaggi incontrati anche per mezzo della tanta odiata facoltà di ingegneria, Mario (linux rules!!!...lo sto usando proprio ora!), Alessandro che mi ha fatto sentire un bravo professore del software libero, il caro buon Joe che non vedo da un casino, il pescarese (troppo ultà) Tommaso (sei in debito con

[3]Un grazie anche a Vingla!!!

[4]Si lo so è il nome può ingannare è per questo che io ero Jaque e tu Jack.

[5]Questo è slang a buon intenditor....

me...sai a cosa mi riferisco :)) e il filosofo, ehm pardon, ingegnere Simone, non so quando ma un giorno dovremo ribeccarci e parlare della vita e della sua essenza.

Con questo penso di aver concluso. Perdonatemi o voi che non ci siete per dimenticanza e adesso comincino le feste.