UNIVERSIDADE NOVA DE LISBOA
Faculdade de Ciências e Tecnologia
Departamento de Informática

# EXPLORATIONS IN REVISED STABLE MODELS - A NEW SEMANTICS FOR LOGIC PROGRAMS

Por
**Alexandre Miguel dos Santos Martins Pinto**

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Inteligência Artificial Aplicada

sob orientação do
Prof. Doutor Luís Moniz Pereira

Lisboa
Fevereiro 2005

# Acknowledgments

I must start by thanking my wife, Graça, for being the most loving, supportive and encouraging woman a man can be with. None of this could have been done without her support. Also I must thank my parents for all they have done for me, and my sisters for their continuous care.

I specially thank Prof. Dr. Luís Moniz Pereira, my supervisor, for lots of reasons which include, but are not limited to: sharing with me his innovative thoughts and excitement, proposing me the challenge to pursue this new and exciting world of open possibilities, his availability for discussions even when he had his schedule already filled up, for introducing me to several researchers within the right timing, for teaching me the good practices of serious scientific research and allowing me to do it with him, and also for his friendship and pleasant company.

I must also thank Prof. Dr. José Júlio Alferes for his most valuable feedback and opinion in so many times, Federico Banti for so many discussions and for the precious time he spent trying to find flaws in my work and giving me his opinion on how to solve them.

Internationally, I thank Nicola Leone. Without his precious help and feedback the Revised Stable Models definition would not have reached its final stage so soon.

I acknowledge the *Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa*, and its *Centro de Inteligência Artificial (CENTRIA)* and also the *Centro de Informática da Universidade Nova de Lisboa* for giving me work conditions.

Sílvia Marina, Anabela Duarte and Sandra Raínha are also present in my thank-you list for helping me with bureaucratic papers and arranging my travels during this last year.

Finally, a word of appreciation must go to Marco Correia, António Pestana and Carlos Delgado for hearing me out talking about my work countless times.

# Abstract

After a brief historical overview, the display of the background assumptions and brief study of some properties of Normal Logic Programs, the core of this thesis concerns the definition and study of the properties of a new 2-valued semantics for Normal Logic Programs: the Revised Stable Models semantics.

The main original contributions of this work are:

- A study of some properties of Classical Models and Minimal Models
- A study of the conditions for the absence of Stable Models in a Normal Logic Program (NLP)
- A study of Odd Loops Over Negation (OLONs) and the properties of Minimal Models of NLPs with OLONs
- A study of Infinitely Long Support Chains (ILSC) and the properties of Minimal Models of NLPs with ILSCs
- The *Reductio ad Absurdum* reasoning principle in 2-valued semantics
- The new notion of Generalized Support which extends the Classical notion of Support by adding a *Reductio ad Absurdum* support
- The definition of Revised Stable Models (RSM) — a new 2-valued semantics for NLPs
- The proofs of the Existence, Relevancy and Cumulativity properties of the RSM semantics
- The identification of the NLPs under which the Stable Models semantics also guarantees Existence, Relevancy and Cumulativity

# Sumário

Depois de uma breve descrição histórica, de estruturar as hipóteses de base e de fazer um breve estudo de algumas das propriedades dos Programas Normais em Lógica, o centro desta tese foca-se na definição e estudo das propriedades de uma nova semântica a 2 valores para Programas Normais em Lógica: os Modelos Estáveis Revistos.

As prinicpais contribuições originais deste trabalho são:

- Um estudo de algumas propriedades dos Modelos Clássicos e dos Modelos Mínimos
- Um estudo das condições para a ausência de Modelos Estáveis num Programa Normal em Lógica (PNL)
- Um estudo de Ciclos Ímpares Sobre Negação (CISN) e sobre as propriedades dos Modelos Mínimos de PNLs com CISNs
- Um estudo de Cadeias de Suporte Infinitamente Longas (CSIL) e das propriedades dos Modelos Mínimos de PNLs com CSILs
- O princípio de raciocínio por *Redução ao Absurdo* numa semântica a 2 valores
- A nova noção de Suporte Generalizado que extende a noção de Suporte Clássico pela adição de suporte por *Redução ao Absurdo*
- A definição dos Modelos Estáveis Revistos (MER) — uma nova semântica a 2 valores para PNLs
- As provas das propriedades de Existência, Relevância e Cumulatividade da semântica MER
- A identificação dos tipos de PNL para os quais a semântica de Modelos Estáveis garante Existência, Relevância e Cumulatividade

# Preface

This Thesis is composed of two major parts: part one consists of an introduction to the subject, laying down the background for the following developments, and casting a bird's eye view over the current State-of-the-Art 2-valued semantics for Normal Logic Programs: the Stable Models Semantics.

In part two we present the definition of our new proposed 2-valued semantics for NLP: the Revised Stable Models; then we show some concrete applications of this new semantics; draw some conclusions and sketch the future work in this line of research.

In chapter 1 a brief introduction to the problem is given along with the main lines of motivation and historical overview.

In chapter 2 we set the basic definitions upon which the rest of the work is settled. Some interesting properties of Normal Logic Programs, their Classical Models and Minimal Models are identified here. Chapter 3 deals with a brief critical overview of Stable Models semantics. Their pros and cons are identified and a few hints of a solution are given.

In chapter 4 we present the definition and properties of Revised Stable Models semantics, and also give sketches of proofs of the properties. The formal proofs are in appendix A.

Chapter 5 presents two practical application of RSM semantics: the EVOLP language which deals with evolving logic programs, and the REWERSE European Project under which prototypes of reactive and evolutionary web-based knowledge bases will be built.

Conclusions and Future lines of work, drawn in chapter 6, close this Thesis. The main directions for future work include extending the RSM semantics to Generalized and to Extended Logic Programs, comparing the RSM semantics against other formalisms, and developing more efficient implementations.

In appendixes A, B, C and D the reader can find the proof of the Theorems, Corollaries, Lemmas, Propositions, etc., presented in previous chapters; examples of NLPs and their respective RSMs; the source code of the implementations of the top-down proof-procedure and the RSM calculator; and finally, the tests and results performed with these implementations.

# Contents

# Part I

# The current State-of-the-Art

# Chapter 1

# Introduction

## 1.1 Historical Overview

Since the beginning of Logic Programming in the early 1970's until today's Answer-Set Programming many theoretical and practical issues have been dealt with. Clark's work on program completion, in the late 1970's, had a great impact and it still is today a major reference.

Later, in the 1980's, Reiter's work [35] on default logic opened new perspectives, which then led to auto-epistemic logic. After the stratification notion had been settled, default logic semantics for logic programs were proposed and soon after the Stable Models (SM) semantics marked the computational logic history. The extension of SM semantics for Extended Logic Programs (programs with explicit negation) led to the development of *Answer-Set Programming*[24].

Meanwhile, 3-valued semantics was also an active field of research which produced the Well-Founded Semantics (WFS) [20] which is the today's standard 3-valued semantics. Several comparative studies between SM semantics and WFS semantics have been developed and their close relationship has been found[17].

Stable Models (SM) semantics [21], the generally accepted standard 2-valued semantics for Normal Logic Programs (NLPs), accepts as Models of a NLP precisely the ones that are intuitively expected. In this sense, SM semantics reflects quite naturally the semantics the programmer intends for the NLP.

However, the Stable Models semantics also has some drawbacks — namely, its lack of enjoying the Existence, Relevancy, and Cumulativity properties. Although most part of the SM community has accepted the lack of these convenient properties as natural, their absence still remain a discomfort for some. Namely, the lack of Relevancy has prevented the development and use of top-down query-oriented proof-procedures.

This was the initial concern that drove the creative process for coming up with a new semantics that extended the Stable Models semantics.

The desired properties of the new semantics should include: being an extension to Stable Models

(in order to keep getting the nice intuitive models the Stable Models semantics gives); guaranteing the Existence of at least one Model for any Normal Logic Program; and also enjoying the Relevance and Cumulativity properties so the use top-down querying proof-procedures and tabling methods for speeding up computations are possible.

All of these driving forces led to the revision of the Stable Models semantics and, ultimately, they give birth, in the mind of my supervisor, Prof. Dr. Luís Moniz Pereira, to the initial intuitive idea and approach towards defining Revised Stable Models (RSMs) [32].

The formal definition of the new Revised Stable Models semantics is the result of a most fruitful research work I have the pleasure to develop along with Prof. Dr. Luís Moniz Pereira.

Throughout the investigation path that lead to the formal proofs of the desired properties of RSMs, several interesting results concerning Classical Models and Minimal Classical Models were found. These results, which, to the best of our knowledge, were previously unknown, are themselves interesting and potentially useful.

# Chapter 2

# Background, Definitions, and Notation

## 2.1 Alphabet, Language and Literals

By an alphabet $\mathcal{A}$ of a language $\mathcal{L}$ we mean a (finite or countably infinite) disjoint set of constants, predicate symbols, and function symbols. In addition, any alphabet is assumed to contain a countably infinite set of distinguished variable symbols. A term over $\mathcal{A}$ is defined recursively as either a variable, a constant or an expression of the form $f(t_1, \ldots, t_n)$ where $f$ is a function symbol of $\mathcal{A}$, and the $t_i$s are terms. An atom over $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_n)$ where $p$ is a predicate symbol of $\mathcal{A}$, and the $t_i$s are terms. A literal is either an atom $A$ or its negation *not* $A$ (sometimes written as $\sim A$. We dub default literals those of the form *not* $A$ — $\sim A$. A term (resp. atom, literal) is called ground if it does not contain variables. The set of all ground terms (resp. atoms) of $\mathcal{A}$ is called the Herbrand universe (resp. base) of $\mathcal{A}$. For short we use $\mathcal{H}$ to denote the Herbrand base of $\mathcal{A}$. A normal logic program is a finite set of rules of the form:

$$H \leftarrow L_1, \ldots, L_n, \ (n \geq 0)$$

where $H$ is an atom and each of the $L_i$s is a literal. In conformity with the standard convention we write rules of the form $H \leftarrow$ also simply as $H$. A normal logic program $P$ is called definite if none of its rules contains default literals. We assume that the alphabet $\mathcal{A}$ used to write a program $P$ consists precisely of all the constants, and predicate and function symbols that explicitly appear in $P$. By Herbrand universe (resp. base) of $P$ we mean the Herbrand universe (resp. base) of $\mathcal{A}$. By grounded version of a normal logic program $P$ we mean the (possibly infinite) set of ground rules obtained from $P$ by substituting in all possible ways each of the variables in $P$ by elements of its Herbrand universe.

These are the base concepts of Alphabet, Language, Herbrand Base, and Normal Logic Program as defined in [5].

In this work we restrict ourselves to Herbrand interpretations and models. Thus, without loss of generality (cf. [33]), we coalesce a normal logic program P with its grounded version.

**Definition 2.1.1.** *Atom*
*We will write $\mathcal{A}$ to denote the set of Atoms.*

**Definition 2.1.2.** *Literal*
*A Literal is defined as an Atom or the* default negation *of an Atom. We will write $\mathcal{L}$ to denote the set of Literals. Hence,*

$\mathcal{L} = \mathcal{A} \cup \{\sim \mathtt{a} : \forall_{\mathtt{a} \in \mathcal{A}}\}$

Additionally, the following auxiliary definitions are also considered throughout this document.

We call *Positive Literal* to an atom of the form $\mathtt{a}$, whereas, oppositely, we name *Default Literal* or *Negative Literal* to the *default negation* — represented by the symbol $\sim$ — of an atom $\mathtt{a}$, thus having the form $\sim \mathtt{a}$.

Formal definitions of the *PosLit* and *NegLit* functions follow.

**Definition 2.1.3.** *Positive Literal — $PosLit(L)$*
*PosLit is a function defined over the set of literals $\mathcal{L}$ unto the binary set $\{\top, \bot\}$ of, respectively, 'true' and 'false' logical values.*

$$PosLit : \mathcal{L} \to \{\top, \bot\}$$

$$PosLit(\mathtt{l}) = \top \Leftarrow \mathtt{l} \textit{ is an atom}$$
$$PosLit(\sim \mathtt{l}) = \bot$$

$$\textit{for all } \mathtt{l} \in \mathcal{L}$$

And also,

**Definition 2.1.4.** *Negative Literal — $NegLit(L)$*
*NegLit is a function defined over the set of literals $\mathcal{L}$ unto the binary set $\{\top, \bot\}$ of, respectively, 'true' and 'false' logical values.*

$$NegLit : \mathcal{L} \to \{\top, \bot\}$$

$$NegLit(\mathtt{l}) = \bot \Leftarrow \mathtt{l} \textit{ is an atom}$$
$$NegLit(\sim \mathtt{l}) = \top$$

$$\textit{for all } \mathtt{l} \in \mathcal{L}$$

Moreover, similarly to what is defined in [7], we also define the *Default Conjugate* of a *Positive Literal* $\mathtt{a}$ to be its correspondent *Negative Literal* $\sim\mathtt{a}$; and conversely, the *Default Conjugate* of a *Negative Literal* $\sim\mathtt{a}$ to be its correspondent *Positive Literal* $\mathtt{a}$.

**Definition 2.1.5.** *Default Conjugate — $conj_D(L)$*

$conj_D : \mathcal{L} \to \mathcal{L}$

$conj_D(\mathtt{l}) = \ \sim \mathtt{l}$
$conj_D(\sim \mathtt{l}) = \mathtt{l}$

*for all* $\mathtt{l} \in \mathcal{L}$

## 2.2 Normal Logic Programs

**Definition 2.2.1.** *Rule*
*We consider a* Rule $\mathtt{r}$ *to be of the form*

$$\mathtt{r} = \mathtt{h} \leftarrow \mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_m \tag{2.1}$$

*where* $n, m \geq 0$*, and the* $\mathtt{h}$*,* $\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n$ *are atoms, and the* $\sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_m$ *are* Negative Literals*.*

*We call* $\mathtt{h}$ *the* head *of the Rule, and the set* $\{\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_m\}$ *the* body *of the Rule.*

*We will use* $\mathcal{R}$ *to denote the set of Rules.*

**Definition 2.2.2.** *Normal Logic Program*
*A* Normal Logic Program*, (NLP for short), is a (possibly infinite) set of Rules as defined in equation 2.1.*

We consider in this text all rules of a NLP to be *Ground Rules*, i.e., without variables. As a NLP can be an infinitely long set of rules there is no loss of generality.

**Definition 2.2.3.** *The* head *function*
*The* head *function is defined over the set of NLP Rules* $\mathcal{R}$*, unto the set of NLP Atoms* $\mathcal{A}$

$$head : \mathcal{R} \to \mathcal{A}$$

*Being* $\mathtt{r}$ *a NLP rule —* $\mathtt{r} \in \mathcal{R}$ *— as defined in equation (2.1), the* head *function is thus defined as*

$$head(\mathtt{r}) = \mathtt{h} \tag{2.2}$$

$head(\mathtt{r})$ *is thus the* Positive Literal *on the left-hand side of the '*$\leftarrow$*' rule symbol.*

**Definition 2.2.4.** *The* Heads *function*
*The* Heads *function is defined over the set of Normal Logic Programs $\mathcal{P}$, unto the set of NLP Atoms $\mathcal{A}$*

$$Heads : \mathcal{P} \rightarrow \mathcal{A}$$

*Being $P$ a NLP, the* Heads *function is thus defined as*

$$Heads(P) = \{\mathtt{h} : \mathtt{h} = head(\mathtt{r}) \wedge \mathtt{r} \in P\} \tag{2.3}$$

*$Heads(P)$ is thus the set of heads of rules of $P$.*

**Definition 2.2.5.** *The* body *function*
*The* body *function is defined over the set of NLP Rules $\mathcal{R}$, unto the set of subsets of NLP Literals $Pow(\mathcal{L})$*

$$body : \mathcal{R} \rightarrow Pow(\mathcal{L})$$

*Being $\mathtt{r}$ a NLP rule — $\mathtt{r} \in \mathcal{R}$ — as defined in equation (2.1), the* body *of a rule $\mathtt{r}$ is then*

$$body(\mathtt{r}) = \{\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_m\} \tag{2.4}$$

*where $n, m \geq 0$.*

*$body(\mathtt{r})$ is thus the set of literals on the right-hand side of the '$\leftarrow$' rule symbol.*

**Definition 2.2.6.** *$Len(P)$ - Length of NLP $P$*
*The* Length of the Normal Logic Program *$P$ is the number of rules of $P$*
*$Len(P) = \#P$*

Naturally, $Len(P)$ can be infinite.

**Definition 2.2.7.** *$DC(P, \mathtt{a})$ - Direct Children of positive literal $\mathtt{a}$ in NLP $P$*
*The set of* Direct Children of literal $\mathtt{a}$ in a Normal Logic Program *$P$, $DC(P, \mathtt{a})$ is defined as*
*$DC(P, \mathtt{a}) = \{\mathtt{l} : \mathtt{l} \in body(\mathtt{r}_i) \wedge \mathtt{r}_i \in P \wedge head(\mathtt{r}_i) = \mathtt{a}\}$*

In this Thesis we will often refer to differences between sets of atoms. For convenience, we will write these differences as $S_1 - S2$ with the same meaning of the traditional notation of $S_1 \setminus S2$.

**Definition 2.2.8.** *$Rel(P, \mathtt{a})$ - Relevant set of rules of NLP $P$ with respect to positive literal $\mathtt{a}$*
*The Relevant set of rules of a Normal Logic Program $P$, with respect to a literal $\mathtt{a}$, $Rel(P, \mathtt{a})$ is defined as*

$$Rel(P, \mathtt{a}) = \{\mathtt{r}_i \in P : head(\mathtt{r}_i) = \mathtt{a}\} \cup \bigcup_{x:\mathtt{r}_i \in P \wedge head(\mathtt{r}_i) = \mathtt{a} \wedge (\mathtt{x} \in body(\mathtt{r}_i) \vee \sim \mathtt{x} \in body(\mathtt{r}_i))} Rel(P, \mathtt{x}) \tag{2.5}$$

Throughout this document we will write $P_a$ as a shorthand notation for $Rel(P, a)$, and $P_{\overline{a}}$ to denote the set of rules of P which are not in $P_a$, i.e., $P_{\overline{a}} = P - P_a$.

Thus, such a $P_a$ and $P_{\overline{a}}$ constitute a partition of $P$

$$(P = P_a \cup P_{\overline{a}}) \wedge (P_{\overline{a}} \cap P_a = \emptyset)$$

**Definition 2.2.9.** *$RelLit(P, \mathtt{a})$ - Set of relevant literals of NLP $P$ with respect to positive literal $\mathtt{a}$*
*The set of relevant literals of a Normal Logic Program $P$, with respect to a literal $\mathtt{a}$, $RelLit(P, \mathtt{a})$ is defined as*

$$RelLit(P, \mathtt{a}) = \{\mathtt{b} : (\mathtt{b} \in body(\mathtt{r}_i) \vee \sim \mathtt{b} \in body(\mathtt{r}_i)) \wedge \mathtt{r}_i \in Rel(P, \mathtt{a})\} \tag{2.6}$$

**Definition 2.2.10.** *$Derivation - Path(P, \mathtt{a}, \mathtt{b})$ — Derivation Path in NLP $P$ from positive literal $\mathtt{a}$ to literal $\mathtt{b}$*
*Consider a sequence $\mathcal{S}$ of literals of $P$ of the form*

$$\mathcal{S} =< \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \ldots, \mathtt{l}_n >$$

*and let $\mathcal{S}(n)$ denote the nth element of the sequence $\mathcal{S}$.*

*We say the $Derivation - Path(P, \mathtt{a}, \mathtt{b})$ is a* Derivation Path in NLP $P$ from literal `a` to literal `b` *iff $Derivation - Path(P, \mathtt{a}, \mathtt{b})$ is one such a sequence $\mathcal{S}$, such that the first element of $\mathcal{S}$ — $\mathcal{S}(1) = \mathtt{l}_1$ — is the literal $\mathtt{a}$, and the last element of $\mathcal{S}$ — $\mathcal{S}(n) = \mathtt{l}_n$ — is the literal $\mathtt{b}$ or its default conjugate $\sim \mathtt{b}$.*

*Moreover, every $i+1$th element of $\mathcal{S}$ — $\mathcal{S}(i+1)$ — is one of the* Direct Children *of $\mathcal{S}(i)$ if $\mathcal{S}(i)$ is a Positive Literal; otherwise $\mathcal{S}(i+1)$ is one of the* Direct Children *of $conj_D(\mathcal{S}(i))$.*

*Formally, a $Derivation - Path(P, \mathtt{a}, \mathtt{b})$ is a sequence $\mathcal{S}$ such that*

$$\mathcal{S} =< \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_n > \wedge \mathcal{S}(1) = \mathtt{l}_1 = \mathtt{a} \wedge \big((\mathcal{S}(n) = \mathtt{l}_n = \mathtt{b}) \vee (\mathcal{S}(n) = \mathtt{l}_n =\sim \mathtt{b})\big) \wedge$$

$$\forall\ 1 \leq i < n \begin{cases} \mathcal{S}(i+1) \in DC(P, \mathcal{S}(i)) \Leftarrow PosLit(\mathcal{S}(i)) \\ \mathcal{S}(i+1) \in DC(P, conj_D(\mathcal{S}(i))) \Leftarrow NegLit(\mathcal{S}(i)) \end{cases}$$

*Naturally, the sequence $\mathcal{S}$ has, at least 2 elements: $\mathtt{a}$ and $\mathtt{b}$. In this Thesis, since we are focusing in Normal Logic Programs (which do not have* Default Literals *in the heads of rules) we will only consider sequences such that their first elements $\mathcal{S}(1)$ are* Positive Literals.

*We say $n$ is the* length *of the $Derivation - Path(P, \mathtt{a}, \mathtt{b})$.*

**Definition 2.2.11.** *$Loop(\mathtt{L}, P, \mathtt{a})$ — Loop $\mathtt{L}$ in NLP $P$ with respect to positive literal $\mathtt{a}$*
*A* Loop `L` in NLP $P$ with respect to literal `a` *is a finite subset of rules $P$ such that there is in `L` exactly one rule with head $\mathtt{a}$ and exactly one rule with $\mathtt{a}$ or $\sim \mathtt{a}$ in the body. All rules in $Loop(\mathtt{L}, P, \mathtt{a})$ have*

*as head an atom present in the body of exactly one rule of* $Loop(L, P, a)$ *— either directly as a* Positive Literal *or through its default negation as a* Negative Literal.

*Formally,*

$$\begin{aligned} Loop(\mathtt{L}, P, \mathtt{a}) \Leftrightarrow \quad & \mathtt{L} \subseteq P \wedge \\ & \exists^1_{r \in L} head(\mathtt{r}) = \mathtt{a} \wedge \\ & \exists^1_{r' \in L}\big((\mathtt{a} \in body(\mathtt{r}')) \vee (\sim \mathtt{a} \in body(\mathtt{r}))\big) \wedge \\ & \forall_{r'' \in L} \exists_{r''' \in L}\big((head(\mathtt{r}'') \in body(\mathtt{r}''')) \vee (conj_D(head(\mathtt{r}'')) \in body(\mathtt{r}'''))\big) \end{aligned}$$

*A* $Loop(\mathtt{L}, P, \mathtt{a})$ *is thus a set of rules of the form*

$$\begin{aligned} & \mathtt{a} \leftarrow \mathtt{L}_1, \mathtt{B}_1 \\ & \mathtt{l}_1 \leftarrow \mathtt{L}_2, \mathtt{B}_2 \\ & \vdots \\ & \mathtt{l}_n \leftarrow \mathtt{A}, \mathtt{B}_n \end{aligned}$$

*where for every* $i \leq n$, $\mathtt{L}_i = \mathtt{l}_i \vee \mathtt{L}_i = \sim \mathtt{l}_i$ *and each* $\mathtt{B}_i = \mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_r, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_s$, *with* $r, s \geq 0$, *and* $\mathtt{A} = \mathtt{a} \vee \mathtt{A} = \sim \mathtt{a}$.

**Proposition 2.2.1.** *In a* $Loop(\mathtt{L}, P, \mathtt{a})$ *there is always exactly one* Derivation-Path(L,a,a).

*Proof.* Immediate from the definition of $Loop(\mathtt{L}, P, \mathtt{a})$. □

**Definition 2.2.12.** *PLoop(PL,P,a) — Positive-Loop in NLP* $P$ *with respect to positive literal* a

*A* Positive Loop in $P$ with respect to literal a *is a* $Loop(\mathtt{PL}, P, \mathtt{a})$ *such that there are no Negative Literals in its* $Derivation - Path(\mathtt{PL}, \mathtt{a}, \mathtt{a})$.

*Formally,*

$$\begin{aligned} PLoop(\mathtt{PL}, P, \mathtt{a}) \Leftrightarrow \quad & \mathtt{PL} \subseteq P \wedge \\ & \exists^1_{r \in PL} head(\mathtt{r}) = \mathtt{a} \wedge \\ & \exists^1_{r' \in PL} \mathtt{a} \in body(\mathtt{r}') \wedge \\ & \forall_{r'' \in PL} \exists_{r''' \in PL} head(\mathtt{r}'') \in body(\mathtt{r}''') \end{aligned}$$

*A* $PLoop(\mathtt{PL}, P, \mathtt{a})$ *is thus a set of rules of the form*

$$\begin{aligned} & \mathtt{a} \leftarrow \mathtt{l}_1, \mathtt{B}_1 \\ & \mathtt{l}_1 \leftarrow \mathtt{l}_2, \mathtt{B}_2 \\ & \vdots \\ & \mathtt{l}_n \leftarrow \mathtt{a}, \mathtt{B}_n \end{aligned}$$

*where for every* $i \leq n$, $\mathtt{B}_i = \mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_r, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_s$, *with* $r, s \geq 0$.

*From the definition it follows immediately that,* $\mathtt{S}$ *being the sequence of* $PLoop(\mathtt{PL}, P, \mathtt{a})$ *with* $n$ *elements, has only* Positive Literals, *i.e.,*

$$\forall_{1 \leq i \leq n} PosLit(\mathtt{S}(i))$$

**Definition 2.2.13.** *LON(NL,P,a) — Loop Over Negation in* $P$ *with respect to positive literal* a

*A* Loop Over Negation in $P$ with respect to literal a *is a* NL *such that* $Loop(\mathtt{NL}, P, \mathtt{a})$ *holds and there is*

*at least one* Negative Literal *in its* $Derivation - Path(\mathtt{NL}, \mathtt{a}, \mathtt{a})$, *and the last literal of the* $Derivation - Path(\mathtt{NL}, \mathtt{a}, \mathtt{a})$ *is a* Negative Literal.

*Formally,*

$$\begin{aligned} LON(\mathtt{NL}, P, \mathtt{a}) \Leftrightarrow \quad & \mathtt{NL} \subseteq P \wedge \\ & \exists^1_{\mathtt{r} \in \mathtt{NL}} head(\mathtt{r}) = \mathtt{a} \wedge \\ & \exists^1_{\mathtt{r}' \in \mathtt{NL}} \sim \mathtt{a} \in body(\mathtt{r}') \wedge \\ & \forall_{\mathtt{r}'' \in \mathtt{NL}} \exists_{\mathtt{r}''' \in NL} (head(\mathtt{r}'') \in body(\mathtt{r}''')) \vee (conj_D(head(\mathtt{r}'')) \in body(\mathtt{r}''')) \end{aligned}$$

*A* $LON(\mathtt{NL}, P, \mathtt{a})$ *is thus a set of rules of the form*

$$\begin{aligned} & \mathtt{a} \leftarrow \mathtt{L}_1, \mathtt{B}_1 \\ & \mathtt{l}_1 \leftarrow \mathtt{L}_2, \mathtt{B}_2 \\ & \vdots \\ & \mathtt{l}_n \leftarrow \sim \mathtt{a}, \mathtt{B}_n \end{aligned}$$

*where for every* $i \leq n$, $\mathtt{L}_i = \mathtt{l}_i \vee \mathtt{L}_i = \sim \mathtt{l}_i$ *and each* $\mathtt{B}_i = \mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_r, \sim \mathtt{c}_1, \sim \mathtt{c}_2, \ldots, \sim \mathtt{c}_s$, *with* $r, s \geq 0$.

*S being the sequence of* $LON(\mathtt{NL}, P, \mathtt{a})$ *with n elements, since* $NegLit(S(n))$ *is true, it follows immediately from the definition that,*

$$(\exists_{1 < i \leq n} NegLit(S(i)))$$

**Definition 2.2.14.** $NNL(\mathtt{LON})$ *— Number of Negative Literals of a* `LON`

*The* Number of Negative Literals of a `LON`, $NNL(\mathtt{LON})$, *is a function defined over the set of* Loops Over Negation $\mathcal{LON}$ *— LONs which are considered to be as defined in 2.2.13 — unto the set of integers* $\mathbb{N}$.

*Let* $\mathtt{L}$ *be a* $LON(\mathtt{L}, P, \mathtt{a})$, *D be the* $Derivation - Path(\mathtt{L}, \mathtt{a}, \mathtt{a})$, *and* $\mathtt{S}$ *be the sequence of* $Derivation - Path(\mathtt{L}, \mathtt{a}, \mathtt{a})$. *Since* $\mathtt{L}$ *is a* `LON`, *we know that* $\mathtt{S} =< \mathtt{a}, \mathtt{l}_1, \mathtt{l}_2, \ldots, \sim \mathtt{a} >$.

*Formally,*

$$NNL : \mathcal{LON} \rightarrow \mathbb{N}$$

$$NNL(S) = \sum_{i=1}^{n} \begin{cases} 1 \Leftarrow NegLit(\mathtt{S}(i)) \\ 0 \Leftarrow PosLit(\mathtt{S}(i)) \end{cases}$$

*It follows trivially from the definition of* $LON$ *that* $NNL(LON) \geq 1$, *for every* $LON$.

**Definition 2.2.15.** $ELON(\mathtt{EL}, P, \mathtt{a})$ *— Even Loop Over Negation in NLP P with respect to positive literal* $\mathtt{a}$

*A* $\mathtt{EL} \subseteq P$ *is an* $ELON(\mathtt{EL}, P, \mathtt{a})$ *iff* $LON(\mathtt{EL}, P, \mathtt{a})$ *holds and* $NNL(\mathtt{EL})$ *is even.*

*Formally,* $ELON(\mathtt{EL}, P, \mathtt{a}) \Leftrightarrow LON(\mathtt{EL}, P, \mathtt{a}) \wedge NNL(\mathtt{EL})$ *is even.*

**Definition 2.2.16.** $OLON(\mathtt{OL}, P, \mathtt{a})$ *— Odd Loop Over Negation in NLP P with respect to positive literal* $\mathtt{a}$

*A* $\mathtt{OL} \subseteq P$ *is an* $OLON(\mathtt{OL}, P, \mathtt{a})$ *iff* $LON(\mathtt{OL}, P, \mathtt{a})$ *holds and* $NNL(\mathtt{OL})$ *is odd.*

*Formally,* $OLON(\mathtt{OL}, P, \mathtt{a}) \Leftrightarrow LON(\mathtt{OL}, P, \mathtt{a}) \wedge NNL(\mathtt{OL})$ *is odd.*

**Definition 2.2.17.** *Atoms Involved in an OLON*
*Let* $\mathtt{L} \subseteq P$*, such that* $OLON(\mathtt{L}, P, \mathtt{a})$ *holds, and* $\mathtt{S}$ *be the sequence of the* $Derivation - Path(\mathtt{L}, \mathtt{a}, \mathtt{a})$ *of* $\mathtt{L}$ *with length* $\mathtt{m}$*.*

*We say that* $\{\mathtt{a}, \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_n\}$ *are the* Atoms Involved in the OLON L *iff*

$$\forall_{1 \leq i \leq n} \exists_{1 \leq j \leq m} \mathcal{S}(j) =\sim \mathtt{l}_i \wedge \exists_{\mathtt{r} \in L} \mathtt{l}_i = head(\mathtt{r})$$

*By definition of OLON,* $\mathtt{a}$ *already satisfies* $\exists_{\mathtt{r} \in L} \mathtt{a} = head(\mathtt{r}) \wedge \exists_{\mathtt{r'} \in L} \sim \mathtt{a} \in body(\mathtt{r'})$*.*

*We write* $AIO(\mathtt{L}, P)$ *to denote the set of* Atoms Involved in the OLON $\mathtt{L}$ *of* $P$*. Hence,*

$$AIO(\mathtt{L}, P) = \{\mathtt{a}, \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_n\}$$

In this sense, any `OLON` in a NLP can be rewritten in such a way that the resulting `OLON'` has exactly $n$ rules, being $n$ the odd number of default literals in the original `OLON`. The resulting OLON has as heads of all rules only the *Atoms Involved in the* original *OLON*.

An example helps to explain the intuitive idea.

*Example 2.2.1.* Rewriting an OLON
Let $P$ be a NLP, `a` a positive literal of $P$, and $\mathtt{L} \subseteq P$ such that $OLON(\mathtt{L}, P, \mathtt{a})$, and `L` =

`a` $\leftarrow$ `b, x,` $\sim$ `y`

`b` $\leftarrow\sim$ `c, z`

`c` $\leftarrow\sim$ `d,` $\sim$ `k`

`d` $\leftarrow\sim$ `a, t`

This OLON `L` can be rewritten as `L'` =

`a` $\leftarrow\sim$ `c, z, x,` $\sim$ `y`

`c` $\leftarrow\sim$ `d,` $\sim$ `k`

`d` $\leftarrow\sim$ `a, t`

And the rule `b` $\leftarrow\sim$ `c, z`, still a rule of $P$, is left out of `L'` — the *Canonical OLON*, as we will call it and formally define it in 2.2.19.

Let us now see the formal definition of the rewriting transformation

**Definition 2.2.18.** *OLON rewriting procedure*
*Let* $L \subseteq P$ *such that* $OLON(L, P, a)$*; the number of rules in* $L$*,* $\#L = n_l$*; the number of default literals in* $L$*,* $NNL(L) = nnl$ *and* $nnl < n_l$*. Let also* $B = \{b : r \in L \wedge head(r) = b \wedge b \notin AIO(L, P)\}$*.*

*Then* $L$ *can be rewritten in the following way:*

- *For every literal* $b \in B$*, let* $r_b$ *be the rule of* $L$ *such that* $head(r_b) = b$*, and let* $r_l$ *be the rule of* $L$ *such that* $b \in body(r_l)$*. Substitute* $b$ *in* $body(r_l)$ *by* $body(r_b)$
- *Remove from* $L$ *all the rules* $r$ *such that* $b \in B$*, keeping those rules in* $P$

*$L'$ is the resulting OLON of the rewriting procedure.*

*Clearly, the resulting OLON $L'$ has exactly the same semantics as the original one $L$. This happens because the rewriting procedure just described is a* unfolding *or* Partial Evaluation *of b as described in [12] and [36].*

*By definition, $L'$ is a set of rules of the form*

$$\begin{aligned}
&\lambda_1 \leftarrow\sim \lambda_2, \Delta_1\\
&\lambda_2 \leftarrow\sim \lambda_3, \Delta_2\\
&\qquad\vdots\\
&\lambda_n \leftarrow\sim \lambda_1, \Delta_n
\end{aligned}$$

*where $\lambda_1 = a$. Trivially, $AIO(L', P) = AIO(L, P) = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$.*

**Definition 2.2.19.** *Canonical OLONs*

*In [15] Stefania Costantini defines OLONs — or* odd-cycles *as called therein — as subsets of Normal Logic Programs which have the form of $L'$ just described in the previous OLON rewriting procedure definition. As just seen, any OLON can be rewritten, preserving its semantics, in the way described. So, without loss of generality, the OLONs we will refer to from now on will be considered to have the form*

$$\begin{aligned}
&OLON =\\
&\lambda_1 \leftarrow\sim \lambda_2, \Delta_1\\
&\lambda_2 \leftarrow\sim \lambda_3, \Delta_2\\
&\qquad\vdots\\
&\lambda_n \leftarrow\sim \lambda_1, \Delta_n
\end{aligned}$$

*where $n$ is an odd number, and $\lambda_1 \ldots \lambda_n$ are distinct atoms. Each $\Delta_i$, $i \leq n$, is a (possibly empty) conjunction $\delta_{i_1} \ldots \delta_{i_h}$ of literals (either positive or negative), where for each $\delta_{i_j}$, $i_j \leq i_h$, $\delta_{i_j} \neq \lambda_i$ and $\delta_{i_j} \neq\sim \lambda_i$.*

*We call the $\lambda_1 \ldots \lambda_n$ the* composing atoms *of the OLON, or the* Atoms Involved in the OLON *as previously stated. $AIO(OLON, P) = \{\lambda_1, \lambda_2 \ldots \lambda_n\}$.*

*We will dub* Canonical OLON *such OLONs with this form. Since all OLONs we will refer to from now on will take the form here described, they will all be* Canonical OLONs*.*

## 2.3 Classical Models

We hereby recall the definition of Classical (Herbrand) Models which will be useful to our further developments in this document.

Being P a Normal Logic Program — we are considering only the First Order Propositional Logic in the context of this Thesis, — a Classical Model $M$ of P is a set of positive literals of P such that every rule '$r$' in P is classically satisfied by $M$, i.e., the head of the rule is true in $M$ and/or the body of the rule is false in $M$ (either by having one positive literal in the body which is not in $M$, or by having a negative literal in the body whose correspondent positive literal is in $M$).

We will use the notation $S(P, r, a)$ to denote that the atom `a` satisfies the rule '$r$' of P. Formally,

**Definition 2.3.1.** *Literal $a$ satisfies rule $r$ in $P$ — $S(P, r, a)$*

$$S(P, r, a) \Leftrightarrow head(r) = a \vee \text{`} \sim a' \in body(r)$$

*We will often say a literal* `a` satisfies rule $r$ by the head *to signify $head(r) = a$; and we will also say a literal* `a` satisfies rule $r$ by the body *to mean $\sim a \in body(r)$.*

Additionally, $SSR(P, a)$ will denote the *Set of Satisfied Rules*, of P, by the atom `a`, which is formally defined by

**Definition 2.3.2.** *Set of Satisfied Rules of $P$ by literal $a$ — $SSR(P, a)$*

$$SSR(P, a) = \{r \in P : S(P, r, a)\}$$

As said before, a rule can also be satisfied if there is an atom '$b$' in $body(r)$ that does not belong to $M$. We shall write $SAL(P, r, M)$ to denote that the rule '$r$' of P is *Satisfied by the Absence of a Literal* in $M$. Formally,

**Definition 2.3.3.** *Rule $r$ of $P$ is Satisfied by the Absence of a Literal in $M$ — $SAL(P, r, M)$*

$$SAL(P, r, M) \Leftrightarrow \exists_{b \in body(r)} b \notin M$$

Also, it will be useful to have the formal notion of the set of rules of P which are satisfied only by some literal `a` in $M$. Those are the rules which are satisfied by `a` and are not satisfied by the absence of any literal in $M$, nor by any other literal in $M$.

Hence, we shall write $SUSR(P, a, M)$ to denote the *Set of Uniquely Satisfied Rules* of P, by the literal `a` in the Model $M$; which formally corresponds to

**Definition 2.3.4.** *Set of Uniquely Satisfied Rules of $P$ by literal $a$ in $M$ — $SUSR(P, a, M)$*

$$SUSR(P, a, M) = \{r \in SSR(P, a) : \neg SAL(P, r, M) \wedge (\nexists_{b \in M} b \neq a \wedge S(P, r, b))\}$$

Also, $SM(P, r, M)$ will denote that the rule '$r$' of P is satisfied by the Model $M$. Formally,

**Definition 2.3.5.** *Rule $r$ of $P$ is satisfied by the Model $M$ — $SM(P, r, M)$*

$$SM(P, r, M) \Leftrightarrow \exists_{a \in M} S(P, r, a) \vee SAL(P, r, M)$$

For the sake of simplicity and shortness in the writing of this Thesis, we will write $CM_P(M)$ to denote that $M$ is a Classical model of P. So, formally,

**Definition 2.3.6.** *$M$ is a Classical Model of $P$ — $CM_P(M)$*

$$CM_P(M) \Leftrightarrow \forall_{r \in P} SM(P, r, M)$$

Also, we will write $CM(P)$ to denote the set of Classical Models of P. Formally,

**Definition 2.3.7.** *Set of Classical Models of $P$ — $CM(P)$*

$$CM(P) = \{M : CM_P(M)\}$$

### 2.3.1 Minimal Classical Models

We also present again the definition of Minimal Classical Models.

A Minimal Classical Model, or just Minimal Model for short, $M$ of a Normal Logic Program P, is a Classical Model such that there is no other Classical Model $M'$ of P which is a proper subset of $M$.

In this text we will write $MM_P(M)$ to denote that $M$ is a Minimal Classical Model of P, so formally,

**Definition 2.3.8.** *$M$ is a Minimal Model of $P$ — $MM_P(M)$*

$$MM_P(M) \Leftrightarrow CM_P(M) \wedge \nexists_{M' \subset M} CM_P(M') \tag{2.7}$$

Also, for convenience, we will use $MM(P)$ to denote the set of Minimal Classical Models of P. Formally,

**Definition 2.3.9.** *Set of Minimal Models of $P$ — $MM(P)$*

$$MM(P) = \{M : MM_P(M)\} \tag{2.8}$$

Another intuitive definition for a Minimal Model $M$ of P is: every atom in $M$ satisfies at least one rule '$r$' of P which is not satisfied by any other means, i.e., it is not *Satisfied by the Absence of a Literal*, nor it is satisfied by any other atom in $M$.

We now show this equivalent alternative definition of Minimal Classical Model.

**Theorem 2.3.1.** *Unique Satisfaction Property of Minimal Classical Models*

*If $M$ is a Minimal Classical Model of $P$ then it is a Classical Model of $P$ and for every atom in $M$ there is at least one rule in $P$ that is satisfied by that atom alone.*

$$MM_P(M) \Rightarrow \Big( CM_P(M) \wedge \Big( \forall_{a \in M} \exists_{r \in P} \big( S(P, r, a) \wedge \neg SAL(P, r, M) \wedge \nexists_{b \in M} b \neq a \wedge S(P, r, b) \big) \Big) \Big)$$

$$MM_P(M) \Rightarrow CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} r \in SUSR(P, a, M) \tag{2.9}$$

*Proof.* By definition, $MM_P(M) \Leftrightarrow CM_P(M) \wedge \nexists_{M' \subset M} CM_P(M')$. So,

$$
\begin{aligned}
&MM_P(M) \Leftrightarrow \\
&CM_P(M) \wedge \forall_{M' \subset M} \neg CM_P(M') \Rightarrow \\
&CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} \neg SM(P, r, M \setminus \{a\}) \Leftrightarrow \\
&CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} \neg SAL(P, r, M \setminus \{a\}) \wedge \nexists_{b \in M \setminus \{a\}} S(P, r, b) \Leftrightarrow \\
&CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} \nexists_{b \in M} b \neq a \wedge S(P, r, b) \wedge \neg SAL(P, r, M) \Leftrightarrow
\end{aligned}
$$

Finally, since

$$CM_P(M) \Leftrightarrow \forall_{r \in P} SM(P, r, M) \Leftrightarrow \forall_{r \in P} \exists_{a \in M} S(P, r, a) \vee SAL(P, r, M)$$

we have

$$
\begin{aligned}
&MM_P(M) \Leftrightarrow CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} \nexists_{b \in M} b \neq a \wedge S(P, r, b) \wedge \neg SAL(P, r, M) \Leftrightarrow \\
&\big(\forall_{r \in P} \exists_{a \in M} S(P, r, a) \vee SAL(P, r, M)\big) \wedge \big(\forall_{a \in M} \exists_{r \in P} \nexists_{b \in M} b \neq a \wedge S(P, r, b) \wedge \neg SAL(P, r, M)\big)
\end{aligned}
$$

which leads us inexorably to the conclusion that $\forall_{a \in M} \exists_{r \in P} S(P, r, a) \wedge \neg SAL(P, r, M) \wedge \nexists_{b \in M} b \neq a \wedge S(P, r, b)$.

From the definition of $SUSR(P, a, M)$ we know that

$$r \in P \wedge S(P, r, a) \wedge \neg SAL(P, r, M) \wedge \nexists_{b \in M} b \neq a \wedge S(P, r, b) \Leftrightarrow r \in SUSR(P, a, M)$$

Thus, we can conclude that

$$MM_P(M) \Rightarrow \Big(CM_P(M) \wedge \Big(\forall_{a \in M} \exists_{r \in P} \big(S(P, r, a) \wedge \neg SAL(P, r, M) \wedge \nexists_{b \in M} b \neq a \wedge S(P, r, b)\big)\Big)\Big)$$

or simply,

$$MM_P(M) \Rightarrow CM_P(M) \wedge \forall_{a \in M} \exists_{r \in P} r \in SUSR(P, a, M)$$

□

**Corollary 2.3.1.** *For every element* $\mathtt{a}$ *of a Minimal Model* $M$, $SUSR(P, a, M) \neq \emptyset$

$$MM_P(M) \Rightarrow \forall_{a \in M} SUSR(P, a, M) \neq \emptyset$$

*Proof.* Trivial from Theorem 2.3.1. □

This property of Minimal Classical Models states that for every atom in $M$ there is at least one rule in P that it satisfies alone. This means that the number of atoms in $M$ is, at most, the number of rules in P. Otherwise there would be an atom in $M$, without which, every rule in P would still be satisfied; thus rendering $M$ non-minimal.

**Proposition 2.3.1.** *If $M$ is a Minimal Model of P, then the number of atoms in $M$ is less or equal to the number of rules of P.*

$$MM_P(M) \Rightarrow \#M \leq \#P$$

*Proof.* From theorem 2.3.1 we know that each atom, alone, in $M$ satisfies, at least, one rule of P. Then the number of rules of P is, at least, the number of atoms of M. □

Although this is a rather trivially easy to prove and understand property, to the best of our knowledge, it has never appeared before in the literature.

## 2.4 The Gelfond-Lifschitz Operator - Γ Operator

In their famous paper [21] about Stable Models, Michael Gelfond and Vladimir Lifschitz defined a procedure to determine if an interpretation — set $M$ of atoms — is a Stable Model of a Normal Logic Program P. This procedure is defined in three steps:

1. Calculate a transformed Normal Logic Program $P'$ by deleting from P all rules having $\sim B$ in the body, where $B \in M$. Then delete from all other rules all the remaining default literals. This step became known as the *Program division* $P/M$

2. Since now the Program $P' = P/M$ is negation-free it has a unique Minimal Herbrand Model, i.e., a Minimal Classical Model which we calculate through the iteration of the Van Emden and Kowalski's $T_P$ operator [18]. The Minimal Classical Model is the *Least Fixed Point of the $T_P$ operator* applied to $P/M$, i.e., $lfp(T_P \uparrow^\omega (P/M))$

3. Check if $M$ equals the calculated Minimal Classical Model of the transformed Program. $M$ is a Stable Model iff $M = lfp(T_P \uparrow^\omega (P/M))$.

The first two steps in this process are designed to, in an intuitive way, calculate the *consequences* of the Program P, assuming true the atoms in $M$ and all the others false, i.e., $lfp(T_P \uparrow^\omega (P/M))$. In this document we will use an alternative (much shorter) notation for $lfp(T_P \uparrow^\omega (P/M))$ — $\Gamma_P(M)$.

For any interpretation $I$, we consider $\Gamma_P^0(I) = I$, and $\Gamma_P^{n+1}(I) = \Gamma_P(\Gamma_P^n(I))$.

### 2.4.1 Rules, bodies and Interpretations

**Definition 2.4.1.** *Interpretation $\Gamma_P$-satisfies body of rule*
*For any given NLP $P$, interpretation $I$, and rule $\boldsymbol{r} \in P$, we say that the body of the rule $\boldsymbol{r}$ is* $\Gamma_P$-satisfied *in the interpretation $I$ — written $I \vdash_{\Gamma_P} body(r)$ — , if and only if every positive atom of the body is a $\Gamma_P$ consequence of the interpretation, and for every negative literal of the body of the rule its correspondent*

*positive literal is not in the interpretation, except possibly if the same positive literal is the head of the rule.*

*Formally,*

$$I \vdash_{\Gamma_P} body(r) \Leftrightarrow \forall_{b \in body(r)} b \in \Gamma_P(I) \wedge \nexists_{\sim c \in body(r)} c \neq head(r) \wedge c \in I$$

*Sometimes we will write* interpretation $I$ *$\Gamma_P$-satisfies $body(r)$ to mean $I \vdash_{\Gamma_P} body(r)$.*

We will also use a slightly weaker version of this satisfaction concept as follows.

**Definition 2.4.2.** *Interpretation satisfies body of rule*
*For any given NLP $P$, interpretation $I$, and rule $r \in P$, we say that the body of the rule 'r' is true in the interpretation $I$ — written $I \vdash body(r)$ — , if and only if every positive atom of the body is in the interpretation, and for every negative literal of the body of the rule its correspondent positive literal is not in the interpretation, except possibly if the same positive literal is the head of the rule.*

*Formally,*

$$I \vdash body(r) \Leftrightarrow \forall_{b \in body(r)} b \in I \wedge \nexists_{\sim c \in body(r)} c \neq head(r) \wedge c \in I$$

*Sometimes we will write* interpretation $I$ *satisfies $body(r)$ to mean $I \vdash body(r)$.*

The unique difference between these two notions of satisfaction is the additional requirement the *$\Gamma_P$-satisfaction* demands that every positive literal of the body of a rule must be, not only in the interpretation, but also a $\Gamma_P$ consequence of it. In the simple *satisfaction* version we just demand that every positive literal must be in the interpretation.

In [14] the author proves that if $M$ is a Model (Classical Model) of a NLP $P$ then $\Gamma_P(M) \subseteq M$. Trivially, this is also true for Minimal Models, since they are Classical Models.

Interestingly, if $M$ is a Minimal Model of $P$ it follows that $\Gamma_P^2(M) \supseteq \Gamma_P(M)$. This result which corresponds to Corollary A.1.1 will reveal itself useful for our developments.

### 2.4.2 The RAA set

As just seen, if $M$ is a Model of NLP $P$ then $M \supseteq \Gamma_P(M)$. For some models $M = \Gamma_P(M)$; with the other models $M \supset \Gamma_P(M)$ holds. In these cases the set difference, i.e. $M \setminus \Gamma_P(M)$, contains just atoms which are not supported by the $\Gamma_P$ operator, under $M$.

We will require, when we present the definition of the new semantics, that all the atoms in $M \setminus \Gamma_P(M)$, for any acceptable $M$, are absolutely necessary. These notions of "acceptable" and "absolutely necessary" will become clear when the definition of the new semantics is presented. For now let us say that those atoms in $M \setminus \Gamma_P(M)$ must come from a *Reductio ad Absurdum* reasoning — and that is why they become "absolutely necessary". We will thus name the $M \setminus \Gamma_P(M)$ set as the $RAA_P(M)$ set.

We will often write $A - B$ as an alternative notation for the set difference $A \setminus B$.

**Definition 2.4.3.** *The $RAA_P(M)$ set*
*Let $P$ be a NLP, $M$ a Minimal Model of $P$. We define the $RAA_P(M)$ set in the following way*

$$RAA_P(M) = M - \Gamma_P(M)$$

*$RAA_P(M)$ is thus the subset of atoms of $M$ which are not classically supported under $M$.*

An interesting aspect of every atom $\mathtt{a}$ of a Minimal Model $M$ of a NLP $P$, which is in $RAA_P(M)$, is that every rule $r \in P$ that is uniquely satisfied by $\mathtt{a}$ is satisfied by the body, i.e., $\sim \mathtt{a} \in body(r)$, or else $\mathtt{a}$ directly depends on another $RAA$ atom $\mathtt{b}$. Formally, this corresponds to the following

**Proposition 2.4.1.** $\forall_{a \in RAA_P(M)} r \in SUSR(P, a, M) \Rightarrow \big( \sim a \in body(r) \vee \exists_{b \in RAA_P(M)} b \in body(r) \big)$

*Proof.* Let $P$ be a NLP, $M$ a Minimal Model of $P$, and $\mathtt{a}$ an atom of $RAA_P(M)$. Since $a \in RAA_P(M)$, by definition of $RAA_P(M)$, we know that $a \in M \wedge a \notin \Gamma_P(M)$. Since $a \in M$ and $M$ is a Minimal Model, by corollary 2.3.1 we know that $SUSR(P, a, M) \neq \emptyset$.

Also, since $a \notin \Gamma_P(M)$ we know that $\forall_{r \in P} head(r) = a \Rightarrow M \nvdash_{\Gamma_P} body(r)$. Which means that $\forall_{r \in P} head(r) = a \Rightarrow SAL(P, r, M) \vee \big(\exists_{c \in M} S(P, r, c) \wedge c \neq a\big) \vee (M \vdash body(r) \wedge M \nvdash_{\Gamma_P} body(r)) \Leftrightarrow \forall_{r \in P} head(r) = a \Rightarrow SAL(P, r, M) \vee \big(\exists_{c \in M} S(P, r, c) \wedge c \neq a\big) \vee \exists_{b \in RAA_P(M)} b \in body(r)$.

If it is the case where $\exists_{c \in M} S(P, r, c)$ and $c \neq a$ or $SAL(P, r, M)$, then we it follows immediately that $r \notin SUSR(P, a, M)$. The other remaining cases are the ones where $\sim a \in body(r)$, and $\exists_{b \in RAA_P(M)} b \in body(r)$.

Hence, by definitions 2.2.7, 2.3.1, 2.3.2, 2.3.4, and 2.3.5 we conclude that
$\forall_{a \in RAA_P(M)} r \in SUSR(P, a, M) \Rightarrow \big( \sim a \in body(r) \vee \exists_{b \in RAA_P(M)} b \in body(r) \big)$ □

### 2.4.3 Particularities of Odd-Loops Over Negation

In [15] the author defines the notion of *Handles* of an OLON. Since that notion will be useful for our developments we rewrite it here for a streamlined reading.

#### Handles of OLONs

Consider $P$ a NLP and $OLON \subseteq P$ such that $OLON$ is a *Canonical OLON*. Let us keep in mind the form of the *Canonical OLON*, as we name it here, which corresponds to the $OLON$ definition of Stefania Costantini in [15], and which we will use from now on:

$OLON = \{R_1, R_2, \ldots, R_n\}$, where
$R_1 = \lambda_1 \leftarrow\sim \lambda_2, \Delta_1$
$R_2 = \lambda_2 \leftarrow\sim \lambda_3, \Delta_2$
$\vdots$
$R_n = \lambda_n \leftarrow\sim \lambda_1, \Delta_n$

### AND Handles

**Definition 2.4.4.** *AND Handles of an OLON*
*The $\Delta_i$'s of the OLON rules, in [15], are referred to as the* AND handles *of the OLON. We will keep this naming here for convenience.*

### OR Handles

**Definition 2.4.5.** *OR Handles of an OLON*
*If there is some rule $R_k \in P \wedge R_k \notin OLON$, where $head(R_k) = head(R_i)$, for some $R_i \in OLON$, and $R_k$ is not part of any OLON in P, and for every literal $l_i \in body(R_k)$ and every $\lambda_j \in AIO(OLON, P)$, $l_i \neq \lambda_j \wedge l_i \neq\sim \lambda_j$; then $body(R_k)$ is called an* OR handle *for the OLON, according to [15].*

*Again, we keep here the same naming convention for the* OR handle.

**Active Handles vs Active OLONs** Consider a NLP $P$ and an $OLON \subseteq P$, as described in definition 2.2.19, with $\Delta_1, \Delta_2, \ldots, \Delta_n$ *AND Handles*.

If any of the $\delta_{i_j} \in \Delta_i$ literals of some rule $R_i$ of $OLON$ is false under some interpretation $I$, $\delta_{i_j}$ is said to be an *active AND handle* of the $OLON$, under $I$.

On the other hand if $R_k \in P$ is such that $body(R_k)$ is an *OR Handle* of the $OLON$ and, under a particular interpretation $I$, if $I \vdash_{\Gamma_P} body(R_k)$ then we say $body(R_k)$ is an *active OR handle* for the $OLON$. Again, both the *active AND handle* and *OR Handle* naming are original from [15].

We will use the expression *Active OLON under I* when referring to an OLON without any *active handles*, either *active AND handles* or *active OR handles*, under interpretation $I$.

Likewise, an *Inactive OLON under I* will mean an OLON with at least one *active handle* under interpretation $I$.

### Minimal Models of OLONs

Consider $P$ a NLP and $OLON \subseteq P$ an *Active OLON under M* such that $MM_P(M)$ holds.

From corollary 2.3.1 we know that every atom $a \in M_{OLON} \subseteq M$, being $M_{OLON}$ a Minimal Model of the OLON, satisfies alone at least one rule of the OLON and, therefore, at least one rule of $P$.

Since the OLON is an *Active OLON under M* — which means that every *AND handle* is not active and that there are no active *OR handles* — the only possible way the rules of the OLON can be satisfied is by having in $M$ some of the $\lambda_i$ literals, such that $\lambda_i \in AIO(OLON, P)$ for every $i \leq n$, where $n$ is the length of the OLON. Let us see how the $\lambda_i$s can contribute to minimally satisfy the OLON rules.

Since the $OLON = \{R_1, R_2, \ldots, R_n\}$ has the form

$R_1 = \lambda_1 \leftarrow\sim \lambda_2, \Delta_1$
$R_2 = \lambda_2 \leftarrow\sim \lambda_3, \Delta_2$
$\vdots$
$R_n = \lambda_n \leftarrow\sim \lambda_1, \Delta_n$

we can see that each $\lambda_i$ satisfies alone the two rules

$\lambda_{i-1} \leftarrow\sim \lambda_i, \Delta_{i-1}$, and

$\lambda_i \leftarrow\sim \lambda_{i+1}, \Delta_i$

$\lambda_i$ satisfies the first rule of this pair because $\sim \lambda_i$ is in its body, and it satisfies the second rule because $\lambda_i$ is its head.

Each literal $\lambda_i$ satisfies 2 rules of the OLON. Since the number of rules $n$ in $OLON$ is odd we know that $\frac{n-1}{2}$ atoms satisfy $n-1$ rules of $OLON$. So, $\frac{n-1}{2}+1 = \frac{n+1}{2}$ atoms satisfy all $n$ rules of $OLON$, and that is the minimal number of $\lambda_i$ atoms which are necessary to satisfy all the OLON's rules.

Taking a closer look at the $OLON$ rules we see that $\lambda_2$ satisfies both the first and second rules; also $\lambda_4$ satisfies the third and fourth rules, and so on. So the set $\{\lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$ satisfies all rules in $OLON$ except the last one. Adding $\lambda_1$ to this set, since $\lambda_1$ satisfies the last rule, we get one possible Minimal Model for $OLON$:

$$M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$$

It is worth noting that every atom in $M_{OLON}$ satisfies 2 rules of $OLON$ alone, except $\lambda_1$, the last atom added. $\lambda_1$ satisfies alone only the last rule of $OLON$. The first rule of $OLON$ — $\lambda_1 \leftarrow\sim \lambda_2, \Delta_1$ — despite being satisfied by $\lambda_1$, was already satisfied by $\lambda_2$.

In this case, we call $\lambda_1$ the *top literal* of the OLON under $M$. The formal definition of the notion of *top literal* follows.

**Definition 2.4.6.** *Top Literal of an OLON*
*Let $P$ be a NLP, $M$ a Minimal Model of $P$, $OLON \subseteq P$ an* Active OLON under $M$, *and $M_{OLON} = \{\lambda_1, \lambda_2, \ldots, \lambda_n\} \subseteq M$ a Minimal Model of $OLON$, where $n \geq 1$ and $\lambda_1 \notin \Gamma_P(M)$.*

*We define the* top literal of the OLON under $M$ *to be $\lambda_1$.*

*Clearly, $\lambda_1 \in M$ since $\lambda_1 \in M_{OLON}$ and $M_{OLON} \subseteq M$. Since $\lambda_1 \notin \Gamma_P(M)$ we can trivially conclude that $\lambda_1 \in RAA_P(M)$.*

Returning to our previous example, the other Minimal Models of the $OLON$ can be found in this manner simply by starting with $\lambda_3$, or $\lambda_4$, or any other $\lambda_i$ as we did here with $\lambda_2$ as an example.

Let us take for now, as an example for a deeper study, the $M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$.

Since $\sim \lambda_{i+1} \in body(R_i)$ for every $i < n$, and $\sim \lambda_1 \in body(R_n)$; under $M_{OLON}$ all the $R_1, R_3, R_5, \ldots, R_n$ will have their bodies false. Likewise, since the OLON is *Active under* $M$, and $M \supseteq M_{OLON}$, all the $R_2, R_4, R_6, \ldots, R_{n-1}$ will have their bodies true under $M_{OLON}$ and so, also under $M$.

This means that all $\lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}$ will be in $\Gamma_P(M)$ but not $\lambda_1$. Since $\lambda_1$ is the *top literal of the OLON* we can conclude, generally, that

**Proposition 2.4.2.** $OLON \subseteq P \wedge MM_P(M) \Rightarrow \big(\lambda_i \in M \wedge \lambda_i \notin \Gamma_P(M) \Leftrightarrow \lambda_i \in RAA_P(M)\big)$, *where $\lambda_i$ is the* top literal of $OLON$

*Proof.* Immediate from the previously stated. □

**Proposition 2.4.3.** *$OLON \subseteq P \wedge MM_P(M) \Rightarrow \forall_{\lambda_i \in AIO(OLON,P) \wedge \lambda_i \in M} \lambda_i \in \Gamma_P(M)$, where every $\lambda_i$ is not the* top literal of the *OLON*

*Proof.* Immediate from the previously stated. □

**OLONs and $\Gamma_P$ circularities**

In [10] the authors define a new semantics for Normal Logic Programs named Stable Class semantics. This semantics always guarantees the existence of a Stable Class (a set $S$ of Interpretations such that $\forall_{I \in S} \exists_{J \in S} J = \Gamma_P(I)$). The authors prove that these Classes are cycles in a graph where the nodes are interpretations and there is an edge from a node $N_i$ to a node $N_j$ iff $N_j = \Gamma_P(N_i)$.

There are some similarities between Stable Class semantics and the Revised Stable Models semantics but the do not coincide.

Let us analyze more thoroughly some of the implications of the previous proposition 2.4.2. Assume $OLON \subseteq P$, $MM_P(M)$ and that some $\lambda_i$ is the *top literal of OLON*. We already know that $\lambda_i \in M$ and that $\lambda_i \notin \Gamma_P(M)$. If the *OLON* is *active under* $M$ then the bodies of all the rules of the *OLON* — except maybe some $\sim \lambda_j$ — are true in $M$.

$M_{OLON}$ being a Minimal Model of *OLON* such that $M \supseteq M_{OLON}$, we have also already seen that all $M_{OLON} - \{\lambda_i\} \subseteq \Gamma_P(M)$.

It is interesting to notice that, under these conditions, the body of the last rule $R_n$ of the OLON is now true under $\Gamma_P(M)$.

**Proposition 2.4.4.** *$\Gamma_P(M) \vdash body(R_n)$, where $R_n \in OLON \subseteq P \wedge MM_P(M) \wedge \sim \lambda_i \in body(R_n)$, where $\lambda_i$ is the* top literal of *OLON*

*Proof.* Under $M$ the body of the last rule $R_n$ of *OLON* was false uniquely due to $\lambda_i \in M$ since the rest of the body of the rule was already true in $M$ because *OLON* is *active under* $M$.

So, under $M - \{\lambda_i\}$, the body of the last rule is now true. Since $\sim \lambda_i, \Delta_n$ is the body of the rule $R_n$, and the OLON is *active under* $M$ — $M \vdash \Delta_n$ — we conclude that $\Delta_n \subseteq \Gamma_P(M)$. Since $\lambda_i \notin \Gamma_P(M)$ it follows that $\Gamma_P(M) \vdash body(R_n)$. □

Since $head(R_n) = \lambda_n$, and from this proposition 2.4.4 $\Gamma_P(M) \vdash body(R_n)$, it follows trivially that $\lambda_n \in \Gamma_P(\Gamma_P(M))$, i.e., $\lambda_n \in \Gamma_P^2(M)$. From Corollary A.1.1 we know that $\Gamma_P^2(M) \supseteq \Gamma_P(M)$.

We already know that a Minimal Model of an OLON, with *top literal* $\lambda_i$, has the form $M_{OLON} = \{\lambda_i, \lambda_{i+1}, \lambda_{i+3}, \lambda_{i+5}, \ldots \lambda_{i+\frac{n-1}{2}}\}$

Now, $\Gamma_P^2(M) \supseteq \{\lambda_{i+1}, \lambda_{i+3}, \lambda_{i+5}, \ldots \lambda_{i+1+n-2}, \lambda_n\}$.

Inductively reiterating the application of the $\Gamma_P$ operator we will obtain $\lambda_i \in \Gamma_P^{n+1}$.

Let use see an example to help clarify the intuitive idea.

*Example 2.4.1.* $\Gamma_P$ circularity of the *top literal of the OLON*
Let $P$ be the NLP

$\lambda_1 \leftarrow\sim \lambda_2$

$\lambda_2 \leftarrow\sim \lambda_3$

$\lambda_3 \leftarrow\sim \lambda_1$

For simplicity, the $\Delta_i$ of the rules in this program are empty, but this does not change the general case since it is guaranteed that the *OLON* is *active under* $M$.

Starting with a Minimal Model $M = \{\lambda_1, \lambda_2\}$ we can see that $\Gamma_P(M) = \{\lambda_2\}$, and so, $\Gamma_P(\Gamma_P(M)) = \Gamma_P^2(M) = \{\lambda_2, \lambda_3\}$.

If we continue to apply the $\Gamma_P$ operator we will obtain: $\Gamma_P^3(M) = \{\lambda_3\}$ ; $\Gamma_P^4(M) = \{\lambda_1, \lambda_3\}$.

As we can see, this OLON has length 3 and it took us $4 = n + 1$ iterations of the $\Gamma_P$ operator to obtain back the *top literal of the OLON under* $M$: $\lambda_1$.

It is also worth noticing that if we kept iterating $\Gamma_P$ we would obtain: $\Gamma_P^5(M) = \{\lambda_1\}$, and $\Gamma_P^6(M) = \{\lambda_1, \lambda_3\} = M$

After $2n$ iterations of the $\Gamma_P$ operator — or if we prefer, after $n$ iterations of the $\Gamma_P^2$ operator — we obtain again all the atoms in the original Minimal Model.

The remarks in this example are not just curiosities and they have deep implications.

**Proposition 2.4.5.** *If $P$ is a NLP, $M$ is a Minimal Model of $P$, $OLON \subseteq P$ is an* active OLON under *$M$ with length $n$ and $\lambda_i$ is the* top literal of the *$OLON$* under *$M$, then $\lambda_i \in \Gamma_P^{n+1}(M)$*

*Proof.* We have already seen that if $M_{OLON} \subseteq M$ is a Minimal Model of an *OLON* and if some $\lambda_i$ is its *top literal* then $\lambda_i \in M \wedge \lambda_i \notin \Gamma_P(M)$ which is equivalent to say that $\lambda_i \in RAA_P(M)$ and this necessarily implies that $\lambda_i \in RAA_P(M_{OLON})$. Moreover, as seen before, $\Gamma_P^2(M)$ is again a Classical Model of $P$ and so all the rules in *OLON* are again satisfied.

The $\Gamma_P$ operator iteration removes from the Model the $\lambda_i$ *top literal of the OLON* and a new iteration adds another *composing literal of the OLON*. Since the *OLON*'s length is finite (otherwise it would not be a Loop, let alone an *Odd Loop*); after $n$ iterations of the $\Gamma_P$ operator, plus the initial one to remove the $\lambda_i$, we will obtain again the initial $\lambda_i$. □

**Proposition 2.4.6.** *If $P$ is a NLP, $M$ is a Minimal Model of $P$ and $OLON \subseteq P$ is an* active OLON under *$M$ with length $n$ and $M_{OLON} \subseteq M$ is a Minimal Model of $OLON$, then $\Gamma_P^{2n}(M) \supseteq M$*

*Proof.* Since each application of the $\Gamma_P$ operator will remove the *top literal of the OLON* and a new iteration will add a new one, we know that the $\Gamma_P^2$ operator substitutes *top literals of the OLON* under the new Model $\Gamma_P^2(M)$. I.e., suppose under $M$ the *top literal of the OLON* is $\lambda_i$. Under $\Gamma_P^2(M)$ the *top literal of the OLON* will be $\lambda_{(i+1)mod(n)}$ — where *mod* stands for the *modulus* operation, i.e., the remainder of the integer division. Since $i = (i + n)mod(n)$, after $n$ iterations of $\Gamma_P^2$ operator we will obtain back a Classical Model which contains the original Minimal Model: $\Gamma_P^{2n}(M) \supseteq M$. □

### 2.4.4 Infinitely long support chains

In [19] François Fages showed that *order-consistent* Normal Logic Programs have Stable Models. According to his paper, an *order-consistent* NLP is one such that the $\leq_+$ and $\leq_-$ relations are well-founded. Fages defined the $\leq_+(\leq_-)$ relation in the following way: $p \leq_+ q (p \leq_- q)$ "if there is a path in the predicate dependency graph from $p$ to $q$ with an even (odd) number of negative edges."

Still in Fages' paper, $\leq_+(\leq_-)$ "is well-founded if there is no infinite decreasing chain $x_0 \leq_+ (\leq_-)x_1 \leq_+ (\leq_-)\ldots$, in particular $\leq_+ (\leq_-)$ must be acyclic to be well-founded."

Also in [19], the author gives an example of a Normal Logic Program with no Odd-Loops Over Negation that still has no Stable Models whatsoever. He's example is as follows

*Example 2.4.2 (Fages [19]).* Let $P =$

$p(X) \leftarrow p(s(X))$

$p(X) \leftarrow\sim p(s(X))$

The grounded version of this program is

$p(0) \leftarrow p(s(0))$

$p(0) \leftarrow\sim p(s(0))$

$p(s(0)) \leftarrow p(s(s(0)))$

$p(s(0)) \leftarrow\sim p(s(s(0)))$

$p(s(s(0))) \leftarrow p(s(s(s(0))))$

$p(s(s(0))) \leftarrow\sim p(s(s(s(0))))$

$\vdots$

Although $P$ has no Odd-Loop Over Negation, its unique Minimal Classical Model — which takes $p$ to be true everywhere — is not a well-supported model, since it is not finitely justified.

$$M = \{p(0), p(s(0)), p(s(s(0))), \ldots\}$$

Each $p(X)$ is supported by an infinitely long chain — $p(s(X)), p(s(s(X))), \ldots$. Since in a Stable Model every atom must be consistently and finitely justified, $P$ has no Stable Models.

According to Fages' claim in [19], the cause of lack of Stable Models in this program is the infinitely long support chain, which is, by its infinity, non-well-founded.

As claimed by the author in [19], in this example's program, the cause of the lack of Stable Models is the non-well-founded (infinitely long) $\leq_-$ relation. In fact, since the support chain is infinitely long, one cannot truly refer to the $\leq_+$ or $\leq_-$ relations as Fages proposed, since they correspond respectively to chains with and even or odd number of default negated literals. In an infinitely long support chain like the one in $p(X) \leftarrow\sim p(s(X))$ there is no even or odd number of negations: there is an infinite ($\omega$, the first limit ordinal) number of negations.

A note is appropriate here. If the unground program consisted only on the first rule $p(X) \leftarrow p(s(X))$ there would be a Stable Model which would be the empty set $\emptyset$. Likewise, if the unground

program consisted only on the second rule $p(X) \leftarrow\sim p(s(X))$ there would be two Stable Models: $\{p(0), p(s(s(0))), p(s(s(s(s(0))))), \ldots\}$ and $\{p(s(0)), p(s(s(s(0)))), p(s(s(s(s(s(0)))))), \ldots\}$.

The formal definition of an infinitely long support chain follows.

**Definition 2.4.7.** *Infinitely Long Support Chain for atom* $\mathtt{a}$ *in* $P$ *—* $ILSC(P, C, a)$
*Let* $P$ *be a NLP,* $M$ *a Minimal Model of* $P$ *and* $\mathtt{a}$ *an atom of* $P$*. We say* $C$ *is an* Infinitely Long Support Chain for $\mathtt{a}$ *—* $ILSC(P, C, a)$ *— iff* $C$ *is an infinite subset of* $Rel(P, a)$*. Formally,*

$$ILSC(P, C, a) \Leftrightarrow C \subseteq Rel(P, a) \wedge \forall_{n \in \mathbb{N}} Len(C) > n$$

*We will often write just ILSC as a shorthand notation for* Infinitely Long Support Chain*.*

Thus, an infinitely long support chain for some atom '$l_1$' must have the following general form

$l_1 \leftarrow L_2, \Delta_1$
$l_2 \leftarrow L_3, \Delta_2$
$\vdots$
$l_n \leftarrow L_{n+1}, \Delta_n$
$\vdots$

where $L_i = l_i \vee L_i =\sim l_i$, for every $i$ and each $\Delta_i$ is an arbitrary conjunction of literals of the form $b_1, b_2, \ldots, b_m, \sim c_1, \sim c_2, \ldots, \sim c_o$, with $m, o \geq 0$.

Analogously to what we did for OLONs, we will say the $\{l_1, l_2, \ldots, l_n, \ldots\}$ atoms are the *atoms involved in the ILSC*. Also, we will say $C \subseteq P$ is an *active ILSC under* $M$ iff $\forall_{r \in C} \Delta_i \subset body(r) \Rightarrow M \vdash \Delta_i$.

## 2.5 Sustainable Sets

In our developments in this Thesis we will need to define a new property for interpretations in Normal Logic Programs — the notion of *Sustainable Set*. We define a *Sustainable Set* — which is an interpretation $S$ for a NLP $P$ — as a set in which every atom $a$ is *true* or *undefined* in the Well-Founded Model of $P$ when considered under the context of all other atoms in $S$, if in turn that subset $S \setminus \{a\}$ is sustainable.

**Definition 2.5.1.** *Sustainable Set*

*We say a set* $S$ *is sustainable in NLP* $P$ *iff any atom* $a$ *in* $S$ *does not "go against" the well-founded consequences of the remaining atoms in* $S$*, whenever,* $S \setminus \{a\}$ *itself is a sustainable set. The empty set by definition is sustainable. Not "going against" means that atom* $a$ *cannot be false in the WFM of* $P \cup S \setminus \{a\}$*, i.e.,* $a$ *is either* true *or* undefined*. That is, it belongs to set* $\Gamma_{P \cup S \setminus \{a\}}(WFM(P \cup S \setminus \{a\}))$*.*

*Formally, we say* $S$ *is sustainable iff*

$\forall_{a \in S} S \setminus \{a\}$ *is sustainable* $\Rightarrow a \in \Gamma_{P \cup S \setminus \{a\}}(WFM(P \cup S \setminus \{a\}))$

*If* $S$ *is empty the condition is trivially true.*

Intuitively, if $S$ is a sustainable set — and a subset of a Minimal Classical Model of $P$ — then it is possible to find a, possibly empty, sequence $\mathcal{S}$ of atoms $s_i$ which constitute a partition of $S$, such that for all $i \geq 0$ all the $s_j$ atoms, with $j > i$, are *true* or *undefined* in the Well-Founded Model of $P \cup \{s_k : k \leq i\}$.

Formally, if $M$ is a Minimal Model of $P$, and $S \subseteq M$ is sustainable then

$\exists_{\mathcal{S}}\ \mathcal{S} =< s_1, s_2, \ldots, s_n > \wedge$
$S = \{s_i : 1 \leq i \leq n\} \wedge$
$\forall_{1 \leq i,j \leq n} i \neq j \Rightarrow s_i \neq s_j \wedge$
$\forall_{0 \leq i < j \leq n} \bigcup_j s_j \subseteq \Gamma_{P \cup \bigcup_i s_i}(WFM(P \cup \bigcup_i s_i))$
$n \geq 0$

It is quite easy to see that $S$ being sustainable it implies the previously stated.

In a constructive manner, we can always start with an empty sequence, calculate the Well-Founded Model of $P$ and then calculate its $\Gamma_P$ consequences — $\Gamma_P(WFM(P))$ — which correspond to the set of *true* or *undefined* atoms of $P$. Of this set it is always possible to choose one atom $\mathtt{a}$ and add it to $P$ as a fact — thus obtaining $P \cup \{\mathtt{a}\}$. We can now reiterate the process and calculate $\Gamma_{P \cup \{\mathtt{a}\}}(WFM(P \cup \{\mathtt{a}\}))$ and again choose a new atom $\mathtt{b} \in \Gamma_{P \cup \{\mathtt{a}\}}(WFM(P \cup \{\mathtt{a}\}))$. The sequence is constructed in this way and becomes $< \mathtt{a}, \mathtt{b}, \ldots >$; and $S = \{\mathtt{a}, \mathtt{b}, \ldots\}$.

The sets constructed in this way, provided that they are subsets of some Minimal Model of $P$, are exactly the *Sustainable* Sets. The requirement for $S$ being a subset of some Minimal Model of $P$ aims at guaranteeing that no *unnecessary* atoms are assumed as *true*. If that were the case it would be possible that adding one such *unnecessary* atom to the program as a fact would render as *false* some other atom of $S$. In this case it would not be possible to construct a sequence as decribed.

A thorough exploration of the properties of *Sustainable* Sets is not within the scope of this Thesis and is considered for possible future work.

## 2.6 Current State-of-the-Art

Over the past few decades the scientific community dedicated to the knowledge representation and reasoning problems has developed a number of different semantics for Logic Programs.

2-valued logics semantics — like the Minimal Models, Clark's Completion [13], and the Stable Models [21], amongst others — strive to achieve the most complete information possible, assigning a truth-value to every atom, if possible.

3-valued logics semantics — like the Well-Founded Semantics — sacrifice complete *true-or-false* information about every atom in favor of some desirable properties that some 2-valued semantics lack. The 3-valued semantics also assign a truth-value to every atom — in some cases the truth-value is *undefined*.

There are also other approaches, including multi-valued logics. Some of these multi-valued logics semantics tend to enter the realms of probabilistic reasoning, fuzzy-logic [25], [26] or other similar domains;

while others consider several logical values due to the nature of the specific problem they are used to solve [8].

In this document, besides laying down the foundations of this new semantics, we consider some possible extensions to it. For now these extensions remain on the 2-valued domain, and a possible extension to a 3-valued setting is considered. Multi-valued extensions to Revised Stable Models could possibly be considered in the future, but for now they remain outside the scope of the current goals.

In this Thesis we show the development of a new 2-valued semantics which is an extension to the Stable Models semantics. This new semantics has some of the most important and convenient properties of one 3-valued semantics: the Well-Founded Semantics. With a clear intent to bridge together both Stable Models (and Answer-Set Programming) and Well-Founded Semantics communities, this new semantics offers an new way to handle the value of atoms which, in a 3-valued setting, would be *undefined*.

We now turn to a brief overview on the State-of-the-Art semantics for Normal Logic Programs.

### 2.6.1 Two-Valued Semantics

There are several 2-valued proposals for semantics of Logic Programs: Minimal Models, Clark's Completion, Stable Models are some of the most well-known examples.

Generally accepted by the scientific community working on 2-valued semantics for Logic Programs as the *de facto* standard, the Stable Models semantics [21] gives exactly the results one intuitively expects. We examine the Stable Models definition in the next chapter.

#### Stable Models

The Stable Models were initially designed for Normal Logic Programs, and as its success spread throughout the scientific community, soon an extension for Extended Logic Programs (with explicit negation) was developed. The name *Answer-Sets semantics* [24] has become associated with the Stable Models semantics for Extended Logic Programs. The Answer Set semantics is specially well-suited for knowledge representation problems including reasoning about actions, planning, diagnosis and preferences.

Today, the Answer-Set Programming (ASP for short) community is large and active. The WASP — Working group on Answer Set Programming [38] — is one of the most productive teams working with the Answer Set semantics.

The ASP semantics, a 2-valued semantics for Extended Logic Programs, like the Stable Models, doesn't deal with self-defeating rules like $\mathtt{a} \leftarrow\!\sim \mathtt{a}$, or other arbitrarily long OLONs or ILSCs. Likewise, the same problems Stable Models semantics suffers, are also present in ASP semantics: lack of guarantee of Existence of a model, lack of Relevancy, and lack of Cumulativity. These will be discussed in the next chapter.

Since the ASP semantics applies to Extended LPs there might arise explicit contradictions, for example:

*Example 2.6.1.* In this ELP '$\sim$' is the default negation and '$-$' is the explicit negation.

$-\mathtt{a} \leftarrow\sim \mathtt{b}$

$\mathtt{a} \leftarrow\sim \mathtt{b}$

$\mathtt{c}$

In this case, since there is no way to prove `b`, it's default negation is true and we conclude both `a` and $-\mathtt{a}$: an explicit contradiction.

In case an explicit contradiction arises, ASP semantics assumes the *Ex Contradictio Quod Libet* principle and, thus, the unique model is the whole Herbrand Base. In this previous example, $c$'s truth value — which should always be *true* — becomes unknown, since the unique model of the program is the whole Herbrand Base.

Several other choices could be made, like using a para-consistent version of ASP that would keep $c$'s truth value *true*, $b$'s truth value *false*, and assign the *unknown* or *undefined* truth value to $a$ and $-a$.

### 2.6.2 Three-Valued Semantics

3-valued semantics usually have an elegant solution for most of the problems the 2-valued semantics have: the *undefined* truth-value. By using a third *unknown*, or *undefined*, or something similar, truth-value, the problem of self-defeating rules like $a \leftarrow\sim a$ is neatly solved.

On the domain of 3-valued semantics, the Well-Founded Semantics is by far the most generally accepted and used.

#### Well-Founded Semantics

We can say the Well-Founded Semantics [20] (WFS for short) is to 3-valued semantics as Stable Models is to 2-valued semantics: they both give as models the ones we intuitively expect. However, the WFS has some highly desirable properties which Stable Models lack; namely, Existence, Relevancy and Cumulativity. All of these properties are enjoyed by the WFS because it deals with self-defeating rules like $\mathtt{a} \leftarrow\sim \mathtt{a}$ and other Odd-Loops Over Negation and Infinitely Long Support Chains. The WFS does so by means of the *undefined* truth-value — which it also uses for Even-Loops Over Negation. Moreover, the WFS has only one Model — the Well-Founded Model (WFM) — whereas the Stable Models can have several models — despite its semantics be the intersection of the several models.

The *positive* or *true* atoms of the Well-Founded Model, defined as the least fixed point of the $\Gamma_P^2$ operator [9] — $\Gamma_P^2(I) = \Gamma_P(\Gamma_P(I))$ — can be computed on a bottom-up fashion starting with $I = \emptyset$.

The WFS has been extended to Logic Programs with explicit negation — the Extended Logic Programs. It became known as WFSX [31] — *Well-Founded Semantics with eXplicit negation.*

Taking the extension process one step further, a para-consistent version of the WFSX — the WFSXp [16] — has also been developed. Under this new semantics, even if an atom and its explicit negation are simultaneously derived, the truth-value of all atoms that do not depend on any of those contradictory ones suffers no harm. Para-consistent Logic Programs [1] has been a fruitful area of research and this could

be an interesting possibility for future work under Revised Stable Models: a para-consistent extension to this new semantics.

# Chapter 3

# A criticism of Stable Models

## 3.1 Definition

According to [21], Michael Gelfond's and Vladimir Lifschitz's Stable Models semantics take as models of a Normal Logic Program any interpretation which is a fixed point of the $\Gamma_P$ operator we described in section 2.4. Formally, an interpretation $I$ for the NLP $P$ is a Stable Model of $P$ iff

$$I = \Gamma_P(I)$$

## 3.2 Pros and Cons of Stable Models

The definition of Stable Models is a rather intuitive one, and it is extremely simple and elegant: a Stable Model $M$ of a NLP $P$ is a set of atoms of $P$ which, when assumed to be true, lead us to conclude what was assumed, through the rules of $P$; i.e., the assumptions fully corroborate themselves. Stable Models informally translate into what could be named a "consistent scenario".

No wonder it is still today a major reference and the *de facto* 2-valued standard semantics for NLPs.

Stable Models are also well-supported models, i.e., any atom in a Stable Model has a consistent finite chain of justification atoms that belong to the same model. The several "well-behaved" properties of Stable Models have been discovered and studied during the last years, even decades. Comparisons to other 2-valued and 3-valued semantics have been performed and studied by many researchers.

Despite all of the research that has taken place around Stable Models, this elegant and effective semantics has also some drawbacks, namely, the lack of guarantee of Existence of at least one model for every Normal Logic Program. A very well known example of a Normal Logic Program with no Stable Models is $\mathtt{a} \leftarrow\!\sim \mathtt{a}$.

There are, of course, a number of reasons why this kind of Normal Logic Programs should not — and does not — have any Stable Model. The most commonly used argument says that this $\mathtt{a} \leftarrow\!\sim \mathtt{a}$

NLP is inconsistent: the truth-value of a literal depending on its own negation is not coherent. However, negative self dependencies, as it is the case in the $\mathtt{a} \leftarrow\!\sim \mathtt{a}$ logic program, can appear indirectly and/or context-dependent, for example:

*Example 3.2.1.* Let $P$ be the following NLP

$P =$

$$\mathtt{a} \leftarrow \mathtt{x}, \sim \mathtt{b}$$
$$\mathtt{b} \leftarrow\!\sim \mathtt{c}$$
$$\mathtt{c} \leftarrow\!\sim \mathtt{a}$$
$$\mathtt{x} \leftarrow\!\sim \mathtt{y}$$
$$\mathtt{y} \leftarrow\!\sim \mathtt{x}$$

In this example a depends indirectly on its own negation: a depends on $\sim$ b, b depends on $\sim$ c and c depends on $\sim$ a. The odd number of negations in the dependency loop creates the inconsistency just as in the $\mathtt{a} \leftarrow\!\sim \mathtt{a}$ case. However, in this example, a's self-dependency is conditioned by the truth value of x, which can be either true or false, depending on y's truth-value.

So we see, it is not always trivial to detect when a literal in a Normal Logic Program is involved in a Loop Over Negation with an Odd number of default negations such as the ones presented here. Moreover, in a scenario where a knowledge base (constructed in the form of a Normal Logic Program) is kept updated by adding new rules and facts, it is hard to guarantee that the result of adding a new rule will not create an Odd Loop Over Negation like the one in the previous example. A process that would keep checking for OLONs every time a new rule was added would become increasingly computationally expensive.

A motivating example can clarify why OLONs, for instance, can be needed to model knowledge, and thus, show why we should to solve them by *Reductio ad Absurdum* reasoning.

*Example 3.2.2.* The president of *Morelandia* is considering invading another country. He reasons thus: if I do not invade them now they are sure to deploy Weapons of Mass Destruction (WMD) sometime; on the other hand, if they shall deploy WMD I should invade them now. This is coded by his analysts as:

$P =$

$$\mathtt{deploy\_WMD} \leftarrow\!\sim \mathtt{invade\_now}$$
$$\mathtt{invade\_now} \leftarrow \mathtt{deploy\_WMD}$$

Let us see how reasoning by *Reductio ad absurdum* is employed here. Assume that invade_now is false. By the first rule of the program we are forced to conclude that deploy_WMD is true; and in this case, by the second rule of the program, we are forced to conclude that invade_now is true — and this explicitly contradicts the first assumption that invade_now was false. Having reached an absurd — a contradiction — we are forced to revise our first assumption to the other possible 2-valued-logic truth-value: invade_now is true.

Under the Stable Models semantics this program has no models. Reasoning by absurdity to solve the

OLON in this program is intuitive and guarantees the model existence. Under the new rSM semantics invasion is warranted by the single model $M = \{\texttt{invade\_now}\}$, and no $WMD$ will be deployed.

Clearly, OLONs can appear in Normal Logic Programs, even if they are not purposely programmed. They can be built incrementally by the result of several updates to a common shared knowledge base, coming from different external data sources; or even by self-updating of the knowledge base.

Truly, one can argument that if an OLON appears in a NLP then there is a fundamental flaw in the program design and this should be reviewed at a deeper level because either there is some error in the knowledge representation, i.e., there are some rules which do not truly represent the intended knowledge; or the knowledge we are trying to represent is inconsistent in itself.

However, taking this approach could lead us to a dangerous pitfall; denying the problem will not eliminate it. Since Knowledge Bases can be updated with new rules, a malicious agent with the intent to breakdown some Knowledge Base service could simply send the following update rule to be added to the service: $\texttt{a} \leftarrow\!\sim \texttt{a}$. In case the service would run under the Stable Models semantics the whole Knowledge Base would no longer have any model: the service would breakdown due to lack of semantics. Of course, elementary OLONs as this one can always be easily detected and rejected; but as seen before, more complicated OLONs can be built up and these can be extremely computationally expensive to detect.

We believe it is easier to allow such OLONs to appear and then deal with them in a newer way which we propose with our new semantics.

A thread of updates (self, or external, or a mixture of both) can lead the knowledge base to have one or more OLONs. Considering the Stable Models semantics, the risk of the resulting knowledge base having no models at all — no semantics — is a real one. In this case, surely the knowledge base has become generally inconsistent, but it is still possible that the truth value of some literals remains consistent. Consider the following program that appeared in [11]:

*Example 3.2.3 (Baral and Subrahmanian).* :

P =

$\texttt{a} \leftarrow\!\sim \texttt{a}$

$\texttt{p}$

This program $P$ has no Stable Models, but still the truth value of $\texttt{p}$ should be undeniably *true* and we should be able to know it from $P$'s models. The Well-Founded Semantics [20] solves this problem with its unique Well-Founded Model which says that $\texttt{p}$ is *true* and $\texttt{a}$ is *undefined*. On the other hand, Stable Models semantics says $P$ has no Stable Models.

However, if we want to stick to a 2-valued semantics there is no current solution to this problem. The closest approach to the 2-valued solution of this problem is given in [11]. We will return to this subject in the next chapter and explore it further.

The lack of guarantee of Existence of a model for every Normal Logic Program is not the only drawback of Stable Models semantics. The lack of the Relevancy property makes it impossible to design and build

a top-down query-oriented proof-procedure, which is a very convenient type of tool for knowledge base querying. Another useful property Stable Models semantics lacks is Cumulativity which allows the use of tabling techniques to speedup computations. These are the main lines of motivation for the development of a new 2-valued semantics.

## 3.3 Motivation

### 3.3.1 Desired Properties

For several years, the theoretical impossibility of development of top-down proof procedures for Stable Models semantics (due to the lack of the Relevancy property) plus the also theoretical impossibility of use of tabling techniques (due to the lack of the Cumulativity property) have represented uncomfortable limitations for practical implementation issues.

More recently, and in the sequence of the Knowledge Base Updates research [3], [4], [6], [22], [23] undertaken by several CENTRIA [29] members, the need of guarantee of Existence of at least one Model no matter the course of updates became more and more important.

The EVOLP (section 5.1) language became one of the main practical platforms benefitting from the guarantee of Existence of a Model. Also, the prototypes that will be implemented inside the recent European REWERSE (section 5.2) project can naturally demand the guarantee of Existence of a Model.

The presence of these three properties — guarantee of Existence of a Model, Relevancy, and Cumulativity — in a 2-valued semantics, became the major driving force for the development of a new semantics.

Stable Models semantics is undeniably one of the most, if not the top most, studied 2-valued semantics, and it has several desirable properties: Stable Models are supported models (every atom in a Stable Model is the head of at least one rule whose body is true in the same Stable Model) and are Minimal Models.

Because of this, the new semantics should also be an extension to the Stable Models semantics in the sense that every Stable Model of a NLP P should also be a Model in the new semantics.

### 3.3.2 The main problems

In [14] it is shown that a program like $\mathtt{a} \leftarrow\!\sim \mathtt{a}$, under Stable Models semantics, has the meaning of $\mathtt{a}$ *ex-or* $\mathtt{a}$, where *ex-or* is the exclusive-or logic connective. From this, it is trivial to understand why the SM semantics gives no meaning — no model — to the $\mathtt{a} \leftarrow\!\sim \mathtt{a}$ program. Moreover, a careful look at the "problems" affecting Stable Models leads us to the conclusion that the major cause of trouble are the OLONs and the ILSCs.

In fact, from the notion that in a Stable Model every atom has a consistent finite support chain of atoms which also belong to the same Stable Model we can easily see that there are two possible ways a NLP P can have no Stable Models at all: for every Minimal Model $M$ of P there is one atom $A$ of $M$

such that either

1. its support chain is inconsistent; or

2. its support chain is infinite

In the first case an inconsistent support chain is one where an atom depends on its own negation (which corresponds to Odd-Loops Over Negation).

In the second case an infinite support chain is one where an atom depends on another, and there are infinitely many atoms in this dependency chain. These ideas were formalized in [15], [17], and [19].

In the next chapter we will see how this new semantics solves both problems — dealing with OLONs and with infinite support chains — to guarantee the Existence of a Model, attain Relevancy and Cumulativity.

# Part II

# Revised Stable Models: a new semantics for Normal Logic Programs

# Chapter 4

# Revised Stable Models

In this chapter we lay the foundations and the definition of the Revised Stable Models semantics. We examine the goals which this semantics pursues and the underlying principles that render those goals achievable. The formal definition of the semantics is then presented and explained.

We then expose the desired properties of the semantics and show how intuitively they stem from the definition of the semantics — formal proofs for the properties can be found in the Appendix A.

After this, a complexity analysis of the semantics is sketched, and we conclude this chapter presenting two implementations for Revised Stable Models: one that calculates RSMs for a NLP, and the other is a RSM meta-interpreter which, taking advantage of the Relevancy property of the RSM semantics, allows the user to pose queries that are solved in a top-down fashion. The tests and results of both implementations are presented.

## 4.1 Goals and Aims

The primary goals that motivated the research that led to the development of Revised Stable Models semantics were, as summarized in the previous chapters, having a 2-valued semantics that: (1) guarantees the Existence of a Model for every NLP, (2) that is Relevant, and (3) Cumulative, and (4) whose Models are Minimal and supported including all the Stable Models. The notion of support, however, needs to be extended in order to be possible to achieve the desired properties. In the subsection 4.2.1 we explain how the notion of support must be extended.

### 4.1.1 Options and Choices

One easy and comfortable solution for dealing with the OLONs problem is by means of a third truth value, as Well-Founded Semantics does. However, if we want to stay in a 2-valued semantics, the truth-value of a literal in an OLON, like

$$\mathtt{a} \leftarrow\!\sim \mathtt{a}$$

must be either *true* or *false*. Dealing with the OLON — or *solving the OLON* as we will often write — forces us to choose a truth value of a.

Clearly, choosing the truth value *false* for a would render $\sim$ a *true* and we would be forced, by the rule a $\leftarrow\!\sim$ a, to conclude that a is true — which would explicitly contradict the initial choice.

The only option left is choosing the truth value *true* for a. In this case $\sim$ a would be *false* and the a $\leftarrow\!\sim$ a could not be used to conclude a. In case there are no more rules with head a and true body, there is no way to conclude that a is true, as first assumed. Traditionally, by Closed World Assumption, the truth value of a would be *default false*, which would also contradict the initial choice. However, in this case, as falsehood comes from a default reasoning; whereas in the previous case (a is assumed *false*), as truth is explicitly proved.

This kind of reasoning by *Reductio ad Absurdum* is the intuitive idea chosen for dealing with OLONs in Revised Stable Models semantics.

## 4.2 Fundamental Underlying Principles

Most generally accepted semantics for NLPs, such as Stable Models Semantics and Well-Founded Semantics, have a common philosophical principle underlying them: a notion of support.

### 4.2.1 Notion of Support

Intuitively, being $P$ a NLP, the traditional support concept says that an atom $a$ is supported in a Model $M$ iff there is at least one rule $R$ in $P$ with head $a$ and true body in $M$. Formally,

**Definition 4.2.1.** *Classical Support — $CSup(P, a, M)$*
*Let $P$ be a NLP, $M$ a Minimal Model of $P$, and $a \in M$.*

*We say $a$ is classically supported in $P$ by $M$ — $CSup(P, a, M)$ — iff*

$$\exists_{r \in P} head(r) = a \wedge M \vdash_{\Gamma_P} body(r)$$

This support notion is an embodiment of the philosophical need for justifiability of beliefs. "True conclusions must be supported, caused, by some premises " [30]. In this sense, Logic Programs try to capture and express the physical causality of matters in the chosen subset of the world by means of logical implications: rules.

There are, however, other more complex reasoning mechanisms that can be used. Mechanisms which go beyond plain deduction: induction of new rules (a kind of learning driven by common patterns found in examples), abduction of hypothesis to find an explanation for a given verified outcome (by choosing

one possible alternative scenario from a range of possibilities), *Reductio ad Absurdum* (RAA) reasoning (by choosing one hypothesis over its negation if the later leads to an explicit contradiction), and many others.

Of these, RAA reasoning provides an extension to the traditional notion of support. By the fact that a specific hypothesis leads, through logically deducted consequences, to an explicit contradiction, we are forced to recognize that the initially assumed hypothesis cannot be true and, thus, conclude the opposite of it.

By RAA reasoning, we must "believe" in $a$ if assuming $\sim \mathtt{a}$ leads to a contradiction. The 'reason' for our belief in $\mathtt{a}$ is due to the explicit contradiction that would arise if we believed in $\sim \mathtt{a}$ instead. $\mathtt{a}$s support is the need to avoid the explicit contradiction.

Clearly, RAA reasoning takes place when $\sim \mathtt{a}$ implies, directly or indirectly, $\mathtt{a}$; and this happens only when there are OLONs in the programs. In fact, that is the essential nature of an OLON: an atom depending on its own negation. In this case we want to extend the notion of support by including an RAA-support to the *top literal of the OLON* under some model $M$, which is the one that depends on its own negation under $M$.

As we already know by proposition 2.4.5, $\lambda_i \in \Gamma_P^{n+1}(M)$ when $M$ is a Minimal Model of $P$ and $\lambda_i \in M$ is the *top literal of* $OLON$, such that $OLON \subseteq P$ is an *active OLON under* $M$. This is the condition for RAA-support of a literal.

Since a logic program can have several OLONs, we slightly relax this condition for RAA-support by simply demanding that $\exists_{\alpha\in\mathbb{N}}\Gamma_P^{\alpha}(M) \ni \lambda_i$. This condition trivially includes the more specific one stated in proposition 2.4.5 and it copes with logic programs having several OLONs.

Now, the generalized notion of support which will be considered throughout the rest of this Thesis is formally defined as follows.

**Definition 4.2.2.** *Generalized Support — $GSup(P, a, M)$*
*Let $P$ be a NLP, $M$ be an interpretation of $P$, and $a$ an atom of $M$. We say that $a \in M$ is a generally supported literal by $M$ in $P$*

$$GSup(P, a, M) \Leftrightarrow \big(\exists_{r\in P} head(r) = a \wedge M \vdash_{\Gamma_P} body(r)\big) \vee \big(\exists_{\alpha\in\mathbb{N}}\Gamma_P^{\alpha}(M) \ni a\big)$$

With this extension to the notion of support, no longer the logical deduction is the only means to find what should be believed; the RAA reasoning coupled with the *tertium non datur* principle — to ensure 2-valuedness — creates an extension to the classical notion of support.

It is precisely this extension of the notion of support — by means of RAA reasoning — that lies at the foundation of the new Revised Stable Models semantics.

In general, RAA reasoning takes place when any contradiction arises (if we are not considering a paraconsistent semantics). The specific subset of RAA reasoning taken by Revised Stable Models consists of assuming an atom $\mathtt{a}$ to be true in a Model $M$, if assuming its default negation $\sim a$ would lead to a self-contradiction, namely getting $\mathtt{a}$ as a logically deduced consequence. Hence, the contradiction detection

in RSM semantics is limited to the case where the contradiction itself involves the assumed hypothesis. The general RAA reasoning would be to revise a hypothesis in case *any* contradiction is detected. Such additional contradictions are not available in NLPs, but only after introduction of explicit or default negation in the heads of rules.

Next, we will see that we will require that every atom must in a model have a Generalized Support. We will also require that every atom in a model must respect the consequences of all other atoms in the model. This corresponds to the notion of *Sustainable Set* 2.5.1.

## 4.3 Definition of Revised Stable Models

In this section we present the formal definition of Revised Stable Models(RSM) and explain it. Next we show how Integrity Constraints can be emulated under RSM semantics and give a hint on how to implement them by a RSM Meta-Interpreter.

### 4.3.1 Definition

Revised Stable Models semantics intends to be a semantics that takes as models of a NLP $P$ all its Stable Models and just the extra Minimal Models necessary to guarantee Existence, Relevancy and Cumulativity. Revised Stable Models semantics just takes Stable Models semantics one step further.

Our new semantics should then consider as models the Minimal ones where each and every atom is Generally Supported, and every non-classically supported atom must be the *top literal of an OLON under that Model*, or depend positively on it.

This intuitive description is captured by the following formal definition

**Definition 4.3.1.** *Revised Stable Models*

*$M$ is a Revised Stable Model of a Normal Logic Program $P$ iff*

1. *$M$ is a Minimal Classical Model of $P$, i.e., $MM_P(M)$*
2. *$\exists_{\alpha \geq 2} \Gamma_P^{\alpha}(M) \supseteq RAA_P(M)$*
3. *$RAA_P(M)$ is sustainable*

*The Revised Stable Models semantics of a Normal Logic Program is the intersection of its models, just as the Stable Models semantics is.*

*We write $RSM_P(M)$ to denote that $M$ is a Revised Stable Model of $P$; and we write $RSMS(P)$ to denote the semantics, according to Revised Stable Models, of the Normal Logic Program $P$. Since we defined this as the intersection of all the Revised Stable Models of the program, we say that an atom `a` belongs to the semantics of the program $P$ iff it belongs to every RSM of $P$. Formally,*

$$RSMS(P) = \bigcap_{RSM_P(M)} M \tag{4.1}$$

$$a \in RSMS(P) \Leftrightarrow \left(\forall_M RSM_P(M) \Rightarrow a \in M\right) \tag{4.2}$$

Next we explain the function and justification of each condition above.

**First condition: $M$ is a Minimal Classical Model of $P$** – Minimality of Models ensures maximal classical supportedness of atoms. It is important to note that Minimality of Models, *per se*, does not ensure supportedness of atoms; it only ensures that supported atoms are in a Model. The following example helps to clarify this.

*Example 4.3.1.* $P =$

$\mathtt{a} \leftarrow\!\sim \mathtt{b}$

This program has two Minimal Models: $M_1 = \{\mathtt{a}\}$ and $M_2 = \{\mathtt{b}\}$. Clearly, there is no reason whatsoever to believe in $\mathtt{b}$, but it still is, nonetheless, a Minimal Model of $P$. The Model we wish to have is just $M_1 = \{\mathtt{a}\}$ because $\mathtt{a}$ is supported (classically) on the absence of a proof for $\mathtt{b}$.

Minimality of Models does not prevent $M_2 = \{\mathtt{b}\}$ to be considered a candidate for a Revised Stable Model. However, it prevents the Classical Model $M_3 = \{\mathtt{a}, \mathtt{b}\}$ from reaching the status of candidate to RSM. Undoubtedly, $M_3 = \{\mathtt{a}, \mathtt{b}\}$ satisfies all the rules in $P$ (the only one, in this case), but since it is not minimal, it is guaranteed that at least one atom of $M_3$ is not supported. Hence, $M_3$ is excluded from the potential RSM candidates by the Minimality of Models condition.

Stable Models (SM) are Classically Supported Minimal Models, and we wish to keep them in Revised Stable Models (RSM) as a special case. This condition does not eliminate any SM from being a candidate RSM since SMs are Minimal Models. The proof that every SM is a RSM can be found in the Appendix A — Theorem 4.4.1.

However, not all RSMs are SMs since NLPs with OLONs or ILSCs are dealt with in RSM semantics by resolving the OLON and the ILSC in favor of the positive atom, as explained in previous sections. Atoms in an infinite support chain may be considered in a RSM if they are absolutely minimally necessary.

RSMs are Generally Supported models. Other Minimal Models which are not Classically Supported may or may not be Revised Stable Models; that depends if they respect the other two conditions of the definition.

*Example 4.3.2.* Need for Minimality and Simple RAA reasoning
Let $P$ be

$\mathtt{a} \leftarrow\!\sim \mathtt{a}$

$\mathtt{b} \leftarrow\!\sim \mathtt{a}$

The only candidate Minimal Model is {a}, since {} and {b} are not Models in the Classical sense and {a, b} is not Minimal. The need for *Reductio ad Absurdum* reasoning comes from the requirement to resolve the OLON – an issue not dealt with in the traditional Stable Model semantics.

In fact, considering the default negation ' $\sim'$ as classical negation '$\neg'$, the above rules of this example program $P$ become

$\mathtt{a} \Leftarrow \neg\mathtt{a}$

$\mathtt{b} \Leftarrow \neg\mathtt{a}$

It is known that, generically, $X \Rightarrow Y$ is equivalent to $\neg X \vee Y$, which allows us to re-write the two rules above as

$\mathtt{a} \vee \neg\neg\mathtt{a}$

$\mathtt{b} \vee \neg\neg\mathtt{a}$

which again are equivalent to

$\mathtt{a} \vee \mathtt{a}$

$\mathtt{b} \vee \mathtt{a}$

and to

$\mathtt{a}$

$\mathtt{b} \vee \mathtt{a}$

As we can see, what our initial rules meant was that 'a must be true', and 'either b or a is true'. It is clear that to solve this in a minimal way it is necessary to consider a as true, and that alone is sufficient. Thus the unique RSM is {a}.

Let us now see how the reasoning by *Reductio ad Absurdum* is employed in this case. Considering the first rule of $P$ $\mathtt{a} \leftarrow\!\sim \mathtt{a}$, and since we are working in a 2-valued logic (*tertium non datur*), we can consider two possible scenarios:

1. assume a is false: in this case $\sim$ a is true and we are forced to conclude the head of the first rule: a is true. This is clearly an explicit contradiction against the assumed hypothesis that a is false

2. assume a is true: in this case $\sim$ a is false and we cannot use the first rule $\mathtt{a} \leftarrow\!\sim \mathtt{a}$ of $P$ to prove a. Since there are no other rules for a we have no way to prove a. Traditionally, by Closed-World Assumption (CWA), we conclude a is false.

The important point in this RAA reasoning is that in the second scenario we get a far more weak inconsistency than the one we get in the first scenario. Notice that in the first scenario we assume a is false and we explicitly conclude a is true — a true contradiction. On the other hand, in the second scenario we assume a as true and we simply do not conclude a is true; there is no evidence for as truth — but then again, there is no explicit conclusion that a is false (since in NLPs there are no default literals, nor explicit negation in the heads of rules), which is a much weaker inconsistency.

So, by RAA reasoning, we conclude that a must be true, and in this case the body of the second rule is false and thus the rule $\mathtt{b} \leftarrow\!\sim \mathtt{a}$ is already satisfied. The only model of this program is {a}.

**Second condition:** $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ – As explained before, each and every atom in a RSM must be Generally Supported, i.e., it either is Classically Supported — $\exists_{r \in P} head(r) = a \wedge M \vdash body(r)$ — or it is supported by RAA reasoning — $\exists_{\alpha \in \mathbb{N}} \Gamma_P^\alpha(M) \ni a$.

Atoms in a RSM that are Classically supported are in $\Gamma_P(M)$. This follows trivially from the notion of Classical Support. So, the only atoms of a RSM that need RAA support are those in $RAA_P(M) = M - \Gamma_P(M)$. This is, in fact, the reason for the naming of the $RAA_P(M)$ set.

As we already seen in proposition 2.4.2, every atom $\lambda_i$ in a Minimal Model $M$ of a NLP $P$ that is the *top literal of an active OLON* complies with $\lambda_i \in M \wedge \lambda_i \notin \Gamma_P(M)$. From this is follows immediately that $\lambda_i \in RAA_P(M)$.

Thus, this second condition of the definition of the RSM semantics ensures that every literal in $RAA_P(M)$ has RAA support.

*Example 4.3.3.* Let $P$ be the following NLP:

$a \leftarrow\sim b$

$b \leftarrow\sim a$

$t \leftarrow a, b$

$k \leftarrow\sim t$

$i \leftarrow\sim k$

$P$ has 4 Minimal Models:

$M_1 = \{a, k\}$, $M_2 = \{a, t, i\}$, $M_3 = \{b, k\}$, $M_4 = \{b, t, i\}$. Their correspondent $\Gamma_P(M_i)$ and $RAA_P(M_i)$ are: $\Gamma_P(M_1) = M_1$ and $RAA_P(M_1) = \emptyset$; $\Gamma_P(M_2) = \{a, i\}$ and $RAA_P(M_2) = \{t\}$; $\Gamma_P(M_3) = M_3$ and $RAA_P(M_3) = \emptyset$; $\Gamma_P(M_4) = \{b, i\}$ and $RAA_P(M_4) = \{t\}$.

Since $M_1$ and $M_3$ are Stable Models they trivially satisfy the second condition of the RSM definition. Let us see now what happens with $M_2$ and $M_4$.

$M_2 = \{a, t, i\}, \Gamma_P(M_2) = \{a, i\}, \Gamma_P^2(M_2) = \{a, k, i\}, \Gamma_P^3(M_2) = \{a, k\}, \Gamma_P^4(M_2) = \{a, k\} = \Gamma_P^3(M_2)$. So, after 3 iterations of $\Gamma_P$ we reach a fixed point, starting from $M_2$. One should note that the atoms in $RAA_P(M_2) = \{t\}$ are not elements of any of $\Gamma_P^i(M_2)$, with $i \geq 1$ — one can guarantee this because $\Gamma_P^3(M_2)$ is a fixed point of the $\Gamma_P$ operator.

The second condition of the definition of RSMs is intended to detect negative self-dependencies of atoms in the RAA set. This is clearly not the case with the atom '$t$': the atom does not depend on itself, let alone depending on its own negation. For this reason '$t$' is not a "legitimate" atom to appear in any RAA set.

The case of $M_4$ is precisely the same as $M_2$.

**Third condition: the $RAA_P(M)$ set is sustainable** — The first two conditions of the RSM definition cope with the guarantee that every atom in a RSM is Generally Supported. There is, however, an additional necessary condition: the atoms in the RAA set must respect that natural stratification of the program. By *respecting* the natural stratification we mean that if we create a sequence $\mathcal{S}$ of atoms

made out of all the atoms in the RAA set, and the order of these atoms in the sequence is in accordance with the strata induced by the program's dependency graph, then every $s_j$ atom of $\mathcal{S}$ must be *true* or *undefined* in the context of $P$ plus all the $s_i$ atoms where $i < j$.

Let us see a clarifying example.

*Example 4.3.4.* Negative dependency on OLON
Let $P$ be the NLP

$a \leftarrow\sim a$

$b \leftarrow\sim a$

$k \leftarrow a, \sim b$

We have two Minimal Models $M_1 = \{a, k\}$ and $M_2 = \{a, b\}$. Clearly `a` is involved in an OLON which is always *active under any interpretation*; hence `a` should be in every RSM. This means that the body of the second rule should always be false and so '$b$' should not be a member of any RSM. Both $M_1$ and $M_2$ respect the second condition of the RSM definition, so a third condition is needed in order for $M_1$ to be the only RSM of $P$. The third condition of the RSM definition ensures this. Let us see why.

$RAA_P(M_1) = M_1 - \Gamma_P(M_1) = \{a, k\} - \emptyset = \{a, k\}$, and $RAA_P(M_2) = M_2 - \Gamma_P(M_2) = \{a, b\} - \emptyset = \{a, b\}$.

In this example we can split the program into two strata: the first has just the rule $a \leftarrow\sim a$, and the second has the two rules $b \leftarrow\sim a$ and $k \leftarrow a, \sim b$. Both RAA sets for $M_1$ and $M_2$ have `a` as an element. The point is that, since `a` is the only atom in the first stratum it must the the first in the sequence. So we add `a` as a fact to P and we get

$a$

$a \leftarrow\sim a$

$b \leftarrow\sim a$

$k \leftarrow a, \sim b$

At this point `a` is now *true* in the Well-Founded Model of $P \cup \{\texttt{a}\}$, and, more importantly, `b` became *false*; whereas `k` became also *true*. This clearly shows that the RAA set of $M_2$ — $\{a, b\}$ — is not *Sustainable*; whereas the RAA set of $M_1$ — $\{a, k\}$ — is.

Every Normal Logic Program has a natural stratification, just like as explained in [34]. We want to make sure that every Revised Stable Model respects this natural stratification of the program and that is the role of the *Sustainability* condition. By requiring that every atom in the RAA set to be *true* or *undefined* in the context of all the others we are really demanding that this atom cannot be *false* when the others are considered *true*. I.e., every atom of the RAA must be consistent (not contradicted, not rendered *false*) by assuming the others as *true*. This proviso is only enforced whenever the subset (the one without the atom we are testing) is itself *Sustainable*.

The intrinsic recursivity in the *Sustainability* notion definition reflects the natural stratification of the program and thus *Sustainable* Sets are the ones respecting the stratification.

Let us see another example with a comprehensive explanation.

*Example 4.3.5.* "Illegal" atom in $RAA_P(M)$
Let $P$ be the NLP

$a \leftarrow\sim a$

$b \leftarrow\sim a$

$c \leftarrow\sim b$

$d \leftarrow\sim c$

This program has two Minimal Models: $M_1 = \{a, c\}$ and $M_2 = \{a, b, d\}$. Intuitively, one would expect only $M_1$ to be the unique RSM — and that is precisely what we get with the RSM definition, due to the third condition.

The first rule, as we have already seen, forces us to conclude that `a` will be in every RSM. With this in mind, the body of the second rule ($b \leftarrow\sim a$) should always be false and '$b$' should never be in any RSM. Reiterating this line of thought, '$c$' should be in every RSM, and '$d$' should not. Thus, the unique RSM should be $M_1$.

Let us see what the first two conditions of the RSM definition have to say about the possibility of both $M_1$ and $M_2$ being RSMs.

Both $M_1$ and $M_2$ are Minimal Models of $P$, and so they comply with the first condition. $\Gamma_P(M_1) = \{c\}$ and so $RAA_P(M_1) = M_1 - \Gamma_P(M_1) = \{a, c\} - \{c\} = \{a\}$, which is the *top literal of the unique active OLON*. $\Gamma^2_P(M_1) = \Gamma_P(\Gamma_P(M_1)) = \Gamma_P(\{c\}) = \{a, b, c\}$. Since $\{a, b, c\} \supseteq \{a\}$, we conclude that $\Gamma^2_P(M_1) \supseteq RAA_P(M_1)$, which complies with the second condition of the definition of RSM.

$\Gamma_P(M_2) = \{d\}$ and so $RAA_P(M_2) = M_2 - \Gamma_P(M_2) = \{a, b, d\} - \{d\} = \{a, b\}$. At this point, it is worth noticing that the natural stratification of the program says we must choose the truth-value for `a` before we do it for `b`; and since, by *Reductio Ad Absurdum* `a` must be *true*, `b` must be *false*, and this is the reason why $M_2$ should not be accepted as a RSM, but that is detected only by the third condition of the definition.

$\Gamma^2_P(M_2) = \Gamma_P(\Gamma_P(M_2)) = \Gamma_P(\{d\}) = \{a, b, c, d\}$. Since $\{a, b, c, d\} \supseteq \{a, b\}$, we conclude that $\Gamma^2_P(M_2) \supseteq RAA_P(M_2)$, which complies with the second condition of the definition of RSM.

As we see, the two first conditions of the definition alone are not enough to perfectly characterize the nature of the elements in the $RAA_P(M)$ set.

We have already seen that $M_1 = \{a, c\}$ and $M_2 = \{a, b, d\}$ are both Minimal Models that respect the second condition of the definition. Let us see now what happens with $M_1$ and $M_2$ under the third condition of the definition of RSM.

$M_1 = \{a, c\}$, $\Gamma_P(M_1) = \{c\}$, $RAA_P(M_1) = \{a\}$. $M_2 = \{a, b, d\}$, $\Gamma_P(M_2) = \{d\}$, $RAA_P(M_2) = \{a, b\}$.

Since $RAA_{M_1} = \{a\}$ is a singleton set it is trivial to see that it is *Sustainable*.

Now, concerning $M_2$ we can see that it is not a *Sustainable* Set because there is one subset of $RAA_P(M_2)$ obtained by removing one of its atoms — $RAA_P(M_2) - \{b\}$ in this case — such that the atom removed (`b`) becomes *false* when the other atoms of the $RAA_P(M_2) - \{b\}$ are assumed *true* in the program.

We can see that by calculating the following set of *true* or *undefined* atoms of $P \cup RAA_P(M_2) - \{b\}$:

$RAA_P(M_2) - \{b\} = \{a, b\} - \{b\} = \{a\}$, hence, $P \cup RAA_P(M_2) - \{b\} = P \cup \{a\}$. We now calculate the *true* atoms of $P \cup \{a\}$ — $WFM(P \cup \{a\}) = \{a, c\}$ — and then the *true* or *undefined* atoms of $P \cup \{a\} = \Gamma_{P\cup\{a\}}(\{a, c\}) = \{a, c\}$.

And now we can see that `b` is not an element of $\{a, c\}$ which are the *true* or *undefined* atoms of $P \cup RAA_P(M_2) - \{b\}$, i.e., `b` is not *respected* (it is rendered *false*) by the other atoms in the $RAA_P(M_2)$; hence $RAA_P(M_2)$ is not *Sustainable*.

Let us take a closer step-by-step look. We add `a` to $P$ as a fact and obtain the following resulting program

$P' =$

$a$

$a \leftarrow\sim a$

$b \leftarrow\sim a$

$c \leftarrow\sim b$

$d \leftarrow\sim c$

The set of positive atoms of the Well-Founded Model of a Normal Logic Program can be calculated by obtaining the minimal fixed-point of the $\Gamma^2_P$ operator [9]. So, starting with the minimal interpretation $\emptyset$ — the empty set — we get

$\Gamma^2_{P'}(\emptyset) = \Gamma_{P'}(\Gamma_{P'}(\emptyset)) = \Gamma_{P'}(\{a, b, c, d\}) = \{a\}$;
$\Gamma^2_{P'}(\{a\}) = \Gamma_{P'}(\Gamma_{P'}(\{a\})) = \Gamma_{P'}(\{a, c, d\}) = \{a, c\}$;
$\Gamma^2_{P'}(\{a, c\}) = \Gamma_{P'}(\Gamma_{P'}(\{a, c\})) = \Gamma_{P'}(\{a, c\}) = \{a, c\}$;

At this stage a fixed-point of the $\Gamma^2_{P'}$ operator has been reached and the $\Gamma^2_{P'}$ iteration stops: $WFP(P' = P \cup \{a\}) = \{a, c\}$. We can now calculate the set of *true* or *undefined* atoms of $P' = P \cup \{a\}$ by calculating $\Gamma_{P'}(WFM(P')) = \Gamma_{P'}(\{a, c\}) = \{a, c\}$.

Let us consider another example showing the opposite case: when a Minimal Model with a non-minimal $RAA_P(M)$ set is a RSM.

*Example 4.3.6.* Revised Stable Model with non-minimal RAA set

Let $P$ be

$a \leftarrow\sim b$

$b \leftarrow\sim a$

$c \leftarrow a, \sim c$

$c \leftarrow b, \sim c$

$d \leftarrow b, \sim d$

There are two Minimal Models for this NLP: $M_1 = \{a, c\}$ and $M_2 = \{b, c, d\}$. They both are Revised Stable Models, but let us see why.

$M_1 = \{a, c\}$, $\Gamma_P(M_1) = \Gamma_P(\{a, c\}) = \{a\}$, $RAA_P(M_1) = M_1 - \Gamma_P(M_1) = \{a, c\} - \{a\} = \{c\}$.

$\Gamma^2_P(M_1) = \Gamma_P(\Gamma_P(M_1)) = \Gamma_P(\{a\}) = \{a, c\}$, so $\Gamma^2_P(M_1) \supseteq RAA_P(M_1)$ — the second condition of the definition of RSM is satisfied by $M_1$. Since $RAA_P(M_1)$ is a singleton set it is trivially *Sustainable*.

Let us look now into $M_2$. $M_2 = \{b, c, d\}$, $\Gamma_P(M_2) = \Gamma_P(\{b, c, d\}) = \{b\}$, $RAA_P(M_2) = M_2 - \Gamma_P(M_2) = \{b, c, d\} - \{b\} = \{c, d\}$.

$\Gamma^2_P(M_2) = \Gamma_P(\Gamma_P(M_2)) = \Gamma_P(\{b\}) = \{b, c, d\}$, so $\Gamma^2_P(M_2) \supseteq RAA_P(M_2)$ — the second condition of the definition of RSM is satisfied by $M_2$.

We can see that the rules for `c` do not depend on `d` and vice-versa; hence, assuming one of them *true* should have absolutely no impact on the truth value of the other, and this is why $RAA_P(M_2)$ is *Sustainable*.

Let us consider adding `c` to $P$ as a fact. We will obtain the following resulting program:

$P' =$

$c$

$a \leftarrow\!\sim b$

$b \leftarrow\!\sim a$

$c \leftarrow a, \sim c$

$c \leftarrow b, \sim c$

$d \leftarrow b, \sim d$

We now calculate the Well-Founded Model of this program by iterating the $\Gamma^2_P$ operator until a minimal fixed-point is reached.

$\Gamma^2_{P'}(\emptyset) = \Gamma_{P'}(\Gamma_{P'}(\emptyset)) = \Gamma_{P'}(\{a, b, c, d\}) = \{c\}$;
$\Gamma^2_{P'}(\{c\}) = \Gamma_{P'}(\Gamma_{P'}(\{c\})) = \Gamma_{P'}(\{a, b, c, d\}) = \{c\}$

A fixed-point has been reached: $WFM(P \cup R_2) = \{c\}$. If we now calculate the set of *true* or *undefined* atoms of $P'$ we get $\Gamma_{P'}(WFM(P')) = \Gamma_{P'}(\{c\}) = \{a, b, c, d\}$. Since `d` is an element of this set we know it remains *true* or *undefined* in the context of assuming `c` as *true*— `d` is *respected* by assuming `c`.

The inverse also holds (`c` is *respected* by assuming `d` as *true*). Hence we conclude that $RAA_P(M_2)$ is *Sustainable* and $M_2$ is a RSM.

### 4.3.2 Integrity Constraints

Integrity Constraints (ICs for short) are a useful programming tool profusely used by the community of logic programming. There are also other computer science related communities that use the generic notion of IC for modelling requirements and ensuring consistency or other demanding.

For the logic programming community, the semantics of an IC is usually the forbidding of some conjunction of literals. Intuitively, its meaning is "this and that must not hold under these conditions".

When using the Stable Models semantics as the underlying platform, ICs are usually written in the form

$$SomeLiteral \leftarrow IC, \sim SomeLiteral$$

As we can see, in this form, ICs are written as OLONs with length 1 (we usually call these *direct OLONs*). The interesting part of this form of writing an IC is that the OLON is only an *active OLON*

*under some Model* $M$ when $M \vdash IC$.

What the programmer intends when writing the IC is to render such Models $M$ *unStable*, so to speak. I.e., the programmer wants to make every Model $M$, such that $M \vdash IC$ holds, a non-Stable Model. And, of course, that can be done by means of an OLON. It does not really matter what the *top literal* of the OLON is, as long as it does not occur anywhere else in the program — except maybe in other ICs as head.

This is an elegant and convenient form of writing ICs, taking advantage from the fact that the SM semantics does not deal with OLON as we have seen before, and so eliminating all the Models that turn *active* any OLON.

On the other hand, it is precisely this double-faced feature (not dealing with OLONs) that turns out to be the cause of lack of guarantee of Existence of a Model for every NLP — for instance, if there where an *active OLON* for each and every Minimal Model of a NLP. It is also the cause of lack of the Relevancy property. Consider this example

*Example 4.3.7.* OLON as IC
$P =$

$c \leftarrow a, \sim c$

$a \leftarrow\sim b$

$b \leftarrow\sim a$

There are two Minimal Models: $M_1 = \{a, c\}$ and $M_2 = \{b\}$. Of these, only $M_2$ is a Stable Model. Under Stable Models semantics the truth value of `a` depends not only on the Relevant part of the dependency graph (which only includes the rule with head `a` and the rule with head '$b$'), but also on the rule with head '$c$' because since that rule is an OLON it acts like an IC. That OLON becomes *active* under a Model with '$a$. Since in every Stable Model every OLON must not be *active*, the Minimal Model $M_1$ is not a Stable Model because it includes `a` — the literal that turns the OLON *active*.

Since Revised Stable Models allows OLONs to be *active* one would at first expect that it is not possible to use ICs under RSM semantics. This is clearly not the case. The programmer can still write ICs as it used to do, in the way just described.

In the example 4.3.7 above, both Minimal Models described are Revised Stable Models. Likewise, in case the programmer would have written the program as

$falsum \leftarrow a, \sim falsum$

$a \leftarrow\sim b$

$b \leftarrow\sim a$

There would still be two RSM: $M_1 = \{a, falsum\}$ and $M_2 = \{b\}$. This is just a syntactic change.

If the programmer uses the literal $falsum$ for the heads of ICs, and wants to reject models with $falsum$, then there is just a small additional work to do: to filter the RSMs by rejecting those that have $falsum$.

On the other hand, since the RSM semantics is Relevant (see subsection 4.4.3), the programmer

can use a top-down proof-procedure to pose queries to the knowledge base. In this case, if one wants only answers to queries that ensure that $falsum$ is not derived, then the programmer just needs to add $\sim falsum$ to the query.

For example, considering the previous program with $falsum$ as the head of the IC, if intended original query was

$$?-a$$

the programmer just has to pose the query

$$?-a, \sim falsum$$

In this case, the top-down proof procedure would just use the Relevant rules of the program $a \leftarrow\sim b$ and $b \leftarrow\sim a$ to find out the truth value for `a`, and it would conclude that `a` can be true. Next, considering `a` true, the top-down proof procedure would try to prove $\sim falsum$. Since $falsum$ is the *top literal of an active OLON under* $\{a\}$, the top-down proof procedure would conclude that $falsum$ is true; and therefore, that $\sim falsum$ is false. Hence, the final answer to the $?-a, \sim falsum$ query would be 'no', thus respecting the programmed IC.

The details of the implemented top-down proof procedure can be found in subsection 4.6.1.

## 4.4 Properties

### 4.4.1 Stable Models Extension

Let $SM_P(M)$ denote the fact that $M$ is a Stable Model of the Normal Logic Program $P$, and let $RSM_P(M)$ denote that $M$ is a Revised Stable Model of the Normal Logic Program $P$.

**Theorem 4.4.1.** *Stable Models Extension*
*Every Stable Model of a NLP $P$ is a Revised Stable Model of $P$*

*Formally,*

$$\forall_M SM_P(M) \Rightarrow RSM_P(M)$$

The proof of this theorem can be found in the Appendix A.

The guarantee that every Stable Model of a NLP is also a Revised Stable Model of it allows the end-user to keep benefitting from the useful properties of Stable Models, whenever they exist.

### 4.4.2 Existence

The Revised Stable Models semantics ensures that every Normal Logic Program has a semantics by guaranteeing that there is always, at least, one Revised Stable Model. In this section we show the

intuitive idea behind this claim. The formal proof for this *Existence* property can be found in the Appendix A.

Let $NLP(P)$ denote the fact that $P$ is a Normal Logic Program, and let $RSM_P(M)$ denote that $M$ is a Revised Stable Model of $P$.

**Theorem 4.4.2.** *Existence — Every Normal Logic Program has, at least, one Revised Stable Model*
*Formally,*

$$\forall_P NLP(P) \Rightarrow \exists_M RSM(P)$$

The formal proof of this theorem can be found in the Appendix A.

The intuitive idea for the proof is the following. Since every SM of a NLP $P$ is also a RSM of it, the need to prove the Existence of a RSM for the NLP boils down to the case where $P$ has no Stable Models at all; otherwise the Existence is trivially proven.

So, considering that $P$ has no Stable Models at all, we already know that for every Minimal Model of $P$ either there is an infinitely long support chain for some atoms, or there are *active OLONs*.

Every NLP $P$ has at least one Minimal Model — that is a known fact, and it ensures that the first condition of the definition is always satisfied by at least one candidate model.

For every Minimal Model $M$ of $P$ we can calculate its correspondent $\Gamma_P(M)$ set, and also its $RAA_P(M) = M - \Gamma_P(M)$. Amongst the several $RAA_P(M)$ sets it is always possible to find at least one which is *Sustainable*— thus satisfying the third condition of the definition.

If there are *active OLONs under some Minimal Model* $M$ — $M$ being one of the Minimal Models with *Sustainable* $RAA_P(M)$ set — then we already know by propositions 2.4.2 and 2.4.5 that $M$ satisfies the second condition of the definition.

If $P$ has ILSCs, and $I$ is the set of all the *atoms involved in the ILSC* which are in $RAA_P(M)$, then we show that $\Gamma_P^2(M) \supseteq I$.

And this finishes the proof for Existence.

### 4.4.3 Relevancy

The Relevancy property is a *conditio sine qua non* for having the possibility of building a top-down query-driven proof-procedure. Without the Relevancy property it is guaranteed that it is not possible the develop such a program — one that can answer queries in a top-down proof fashion.

This was one of the major concerns that were in the origin of Revised Stable Models: the definition of a new semantics that extended Stable Models and that enjoyed the Relevancy property.

Let us recall here, for a clearer reading, the definition of the Relevancy property.

*Property 4.4.1.* Relevancy

Let $Sem(P)$ denote the semantics of the NLP $P$. Considering the Revised Stable Models, the semantics of a NLP $P$ is the *intersection* of its models, just as explained in definition 4.3.1.

We say that *A directly depends on B if B occurs in the body of some rule with head A.* Also *A depends on B if A directly depends on B or there is a C such that A directly depends on C and C depends on B.*

Formally,

$$DirectlyDepends_P(A,B) \Leftrightarrow \quad (\exists_{r \in P} head(r) = A \wedge B \in body(r))$$

$$\begin{aligned} Depends_P(A,B) \Leftrightarrow \quad & DirectlyDepends_P(A,B) \vee \\ & (\exists_C DirectlyDepends_P(A,C) \wedge \\ & Depends_P(C,B)) \end{aligned}$$

Let also $Rel(P,A)$ denote the subset of all rules of P whose head is A, or some B on which A depends on. Formally,

$$Rel(P,A) = \{r \in P : (head(r) = A) \vee (head(r) = B \wedge Depends_P(A,B))\}$$

This definition is equivalent to that of 2.2.8.

**Definition 4.4.1.** *Relevancy A semantics is said to be Relevant — or to enjoy the Relevancy property — iff for every program P*

$$A \in Sem(P) \Leftrightarrow A \in Sem(Rel(P,A))$$

In the case of Revised Stable Models, by equations 4.1 and 4.2 we know that, in order for the RSM semantics to be Relevant

$$(\forall_M RSM_P(M) \Rightarrow a \in M) \Leftrightarrow (\forall_{M_a} RSM_{Rel(P,a)}(M_a) \Rightarrow a \in M_a)$$

must hold.

**Theorem 4.4.3.** *The Revised Stable Models semantics is Relevant*

$$(\forall_M RSM_P(M) \Rightarrow a \in M) \Leftrightarrow (\forall_{M_a} RSM_{Rel(P,a)}(M_a) \Rightarrow a \in M_a)$$

The proof for this theorem can be found in the Appendix A.

### 4.4.4 Cumulativity

Cumulativity is the formal property of a semantics which is ultimately the guarantee that the use of tabling/memoizing techniques is allowed.

Intuitively, a Cumulative semantics is one such that it is guaranteed that the atoms which are true in every model of the semantics, remain true even after adding as fact some other true atom in all models. This corresponds to the storing of intermediate lemmas. If one has derived some literal, and found it to be true in every model then, if adding it as a fact to the program causes no change in the truth value of other atoms, then we say the semantics is Cumulative.

**Definition 4.4.2.** *Cumulativity A semantics is said to be Cumulative — or to enjoy the Cumulativity property — iff for every program P*

$$A \in Sem(P) \wedge B \in Sem(P) \Leftrightarrow B \in Sem(P \cup \{A\})$$

**Theorem 4.4.4.** *The Revised Stable Models semantics is Cumulative*

$$\forall_P NLP(P) \Rightarrow \left(\forall_{a,b} a \in RSM(P) \wedge b \in RSM(P) \Rightarrow b \in RSM(P \cup \{a\})\right)$$

The proof for this theorem can be found in the Appendix A.

### 4.4.5 Special-Case Properties of Stable Models

We have seen that the Revised Stable Model semantics is an extension to the Stable Models semantics, in the sense that every Stable Model of a NLP is also a RSM of the same NLP.

Keeping this is mind, the interesting properties of the Revised Stable Models — namely guarantee of Existence of a Model, Relevancy and Cumulativity — turn out to be useful somehow extensible results about the Stable Models semantics itself. Let us see why.

Whenever a Normal Logic Program has no *active OLONs under any Minimal Model* and no *active ILSCs under any Minimal Model* there is no need for Generalized Support for any atom. In this case, all the RSMs are the SMs of the program: the two semantics coincide. This means that, in these special cases where there are no *active OLONs* and no *active ILSCs* under whichever Minimal Models, the Stable Models semantics guarantees the Existence of a Model, is Relevant and Cumulative — the same properties the Revised Stable Models enjoys.

The difference, in what these useful properties are concerned, between SM semantics and RSM semantics, is that RSM semantics always enjoys the Existence, Relevancy and Cumulativity properties no matter what the NLP is; whereas the SM semantics only enjoys these properties when the NLP has no *active OLONs* and no *active ILSCs* no matter under what Minimal Model.

These results — about the Existence, Relevancy and Cumulativity properties — about the Stable Models semantics, to the best of our knowledge, have never before appeared in the literature. Even if the usefulness of the Revised Stable Models semantics is questioned, or its need or motivation considered minor, these results about the properties of Stable Models semantics, made possible only due to the research that involved the Revised Stable Models semantics, are themselves a contribute to a better understanding of the current *de facto* standard in 2-valued semantics for NLPs.

This means that, if one can ensure that his/hers NLP have no OLONs and no ILSCs whatsoever, then, using the Stable Models semantics, it is possible to design and implement a top-down proof-procedure capable of solving queries *a la* Prolog, because in those cases the Relevancy is ensured.

Also, in the same scenario where the absence of OLONs and ILSCs is assured, one can use tabling/memoizing techniques under the SM semantics, because Cumulativity is granted. Moreover, one is sure that there

is always at least one Stable Model if the NLP has no OLONs and no ILSCs.

These results about the SM semantics open new doors to practical implementations of SM-based systems, as long as absence of OLONs and ILSCs is assured.

## 4.5 Complexity Analysis

A profound complexity analysis of this semantics has not yet been performed and it is one of the subjects for future work.

However, a brief glance over the conditions of the definition of the Revised Stable Models semantics allows us to take some safe conclusions: since the minimality of models is required — and this is a typically NP problem — the complexity of the semantics should not be under NP.

The other conditions of the definition essentially involve iterations of the $\Gamma_P$ operator — either to determine if $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ or to check if the RAA set is *Sustainable* (the Well-Founded Model can be calculated by obtaining the least fixed point of the $\Gamma_P^2$ operator).

Each $\Gamma_P$ iteration should be at most polynomial, so for now our conjecture is that the Revised Stable Models semantics is at least NP-complete.

*Conjecture 4.5.1.* The complexity of the Revised Stable Models semantics is at least NP-complete.

## 4.6 Implementation

### 4.6.1 Implementing a RSM Meta-Interpreter

A RSM Meta-Interpreter, as we conceived it, consists in a program that allows the user to load his/hers knowledge base written in the form of a grounded Normal Logic Program. After the loading of the KB is done, the user can then pose queries to the Meta-Interpreter which uses the underlying RSM semantics.

This Meta-Interpreter answers the queries in a top-down fashion — taking advantage of the Relevance property of the RSM semantics. In order to avoid entering in infinite loops, the RSM Meta-Interpreter has a built-in mechanism for detecting and dealing with loops.

#### Dealing with Loops

**Loop Detection** The loop detection mechanism in the RSM Meta-Interpreter consists basically of a list of ancestor literals in the query derivation path, from its root to the current literal. If the literal currently under proof is already in the ancestor list (either positively or negatively) a loop occurs and this situation is detected.

In this case, the RSM Meta-Interpreter will no longer obtain a rule for the literal it is trying to prove — for that would launch the Meta-Interpreter in an infinite loop of proofs.

Instead, the RSM Meta-Interpreter first classifies the loop, and the resolves it. Loop detection and resolution is performed in a *bottom-up* fashion, i.e., when solving a query the meta-interpreter goes down the dependency graph. At some point it may detect a loop which is classified and resolved. The resolution of the loop is then propagated upwards in the dependency graph; hence every loop lyes at the bottom of the query graph. This mechanism of solving the loops and propagating their resolutions *bottom-up* is in close relation to the *Sustainability* condition of the definition — the resolution of the loop is considered *true* and used in the evaluation of the truth-value of the literals above the loop in the query graph, thereby respecting the *Sustainability*.

**Loop Classification** A loop is classified as Positive Loop if the literal being proved is positive, and all the literals in between them are also positive; i.e., there is not one single default negation between the first occurrence of the literal and the second (including the later).

A loop is classified as Even (Odd) Loop if there is an even (odd) number of default negated literals in between the first occurrence of the literal in the ancestors list and the last. If the first occurrence of the literal is itself default negated that one negation does not count for the determination of the "even(odd)-hood" of default literals.

**Loop Resolution** Positive Loops are always solved by assuming the literal in the loop to be false — there is no reason to believe `a` is true if we have, for instance, $a \leftarrow a$. On the other hand, Even (Odd) Loops are solved by assuming the literal in the loop is true (false) — for this will lead, through the even (odd) number of negations upwards, to the conclusion that the literal is true. In the case of Even Loops that is a consistent result: assuming the literal true we conclude it is true. On the other hand, for Odd Loops, since we want to provide RAA support for atoms in Odd Loops, we want them to be concluded true. For that to happen, since the literal is in an Odd Loop, it must be assumed to be false so the odd number of negations between itself and its equal ancestor toggles the truth value to true.

### Keeping Consistency

Throughout the derivation process, the RSM Meta-Interpreter keeps all the conclusions it derived and the assumptions it made for resolving Even Loops in a list of Current Context. Besides being a mechanism for ensuring that future choices for Even Loops are consistent with the other choices already made, this Current Context list serves also as a kind of tabling since it stores the intermediate results it derives.

The literals in this Current Context list are not added to the program as facts, they are just kept in memory to enforce consistency. A positive side-effect of this list is that it reduces the computation time in case we try to prove some literal that has already been proven, or that has been assumed to be false.

When the Meta-Interpreter is asked to try to prove a literal, one of the first things it does is to check in the Current Context list if it was already derived, or its negation assumed. In either case, the computation for that literal stops and the process continues to the next part of the query.

#### Capabilities and Limitations

This first implementation of the RSM Meta-Interpreter is prepared only to deal with grounded NLPs, and its design was with the aim of developing a Rapid Prototype that would show the expected behavior, considering the Revised Stable Models semantics. As said in the Future Work section, a more powerful and efficient implementation of this RSM engine is one of the priorities.

#### Using Integrity Constraints

The simulation and use of Integrity Constraints has already been depicted in subsection 4.3.2. It is up to the programmer to add such rules and tailor the queries so as to include the $\sim falsum$ or $\sim no\_good$, or some other literal of choice for enforcing the respect of the Integrity Constraints.

### 4.6.2 Implementation of a Revised Stable Models calculator

The RSM Calculator implementation takes an approach different from the RSM Meta-Interpreter. We are now interested in a program that, after loading the user's knowledge base in the form of a grounded NLP, calculates all the RSMs of that knowledge base.

Clearly, the process of calculating all the RSMs of a given NLP is a much more computationally expensive task than the simple query-answering one.

Like the RSM Meta-Interpreter, this RSM Calculator was developed with the aim of having a basic mechanism for calculating RSMs; there was no concern for efficiency at this stage. Since the semantics in itself has a high degree of complexity, the process of calculating all the RSMs of a NLP is quite slow. Therefore, we used only small NLPs for the tests. However, we believe the test set used pin-point touches the crucial issues.

Moreover, the implementation of the RSM Calculator follows precisely the formal definition of the RSM semantics: first the RSM Calculator identifies all the Minimal Models of the NLP, then excludes all those that do not respect the second and the third conditions of the definition. This is done precisely as written in the definition: calculating the RAA set for each Minimal Model, checking if $\exists_{\alpha \geq 2} \Gamma_P^{\alpha}(M) \supseteq RAA_P(M)$, and then check if the RAA set is *Sustainable*.

Thus, this implementation of the RSM Calculator turns out to have built-in a series of mechanisms used in deriving consequences of NLP, namely: a $T_P$ operator, a $\Gamma_P$ operator, a $\Gamma_P^2$ operator, the detection of fixed points of $T_P$, $\Gamma_P$, and $\Gamma_P^2$, and also the calculus of the Well-Founded Model of a NLP, and a *Sustainability* checker.

# Chapter 5

# Applications

Many kinds of knowledge-oriented applications can benefit from this new semantics. For one, applications managing knowledge bases usually have to deal with knowledge updates. The more complex these knowledge updates become, the more flexible the underlying semantics has to be.

The 2-valued Stable Models semantics does not always guarantee the existence of a semantics for whichever knowledge base. The continuous update of a knowledge base can lead it to a state where, under Stable Models, it has no semantics, no model. This is an extreme case we seriously want to avoid.

We now examine two scenarios under which the knowledge base evolution is a constant issue. Naturally, the concern about the guarantee of existence of semantics on these scenarios is of paramount importance.

## 5.1 EVOLP

The new EVOLP [3] language provides a simple yet powerful means for programming software agents which evolve over time. The simple and natural way both external updates as well as self-updates are programmed are the main innovations of this language. Naturally, in such a setting one desires to be sure that his/her agent always has a semantics, a model.

The EVOLP language extends common Prolog with a new kind of *assert*, besides *not*s in the heads. The argument of the new *assert* can be any arbitrarily complex rule, including another nested *assert*. An example,

*Example 5.1.1.* A typical EVOLP program

$a \leftarrow not\ b.$

$not\ c \leftarrow not\ a.$

$assert(assert(b \leftarrow not\ a) \leftarrow not\ c) \leftarrow not\ assert(b \leftarrow assert(c)).$

The semantics of EVOLP programs is defined in [3] and we will not enter into details about it here.

The point that matters to this dissertation, though, is closely related to the semantics of a generic evolving knowledge-base; and an EVOLP program is just a particular case of that scenario.

Consider, for instance, the following EVOLP program

*Example 5.1.2.* EVOLP program with camouflaged OLON

$a \leftarrow not\ b.$

$assert(b \leftarrow not\ c) \leftarrow not\ d.$

$assert(assert(c \leftarrow not\ a)) \leftarrow not\ e.$

Let us consider the case where we use the Stable Models semantics as the underlying semantics for this EVOLP program.

Initially, the only Stable Model of this program is $\{a, assert(b \leftarrow not\ c), assert(assert(c \leftarrow not\ a))\}$, since none of $\{b, d, e\}$ holds. EVOLP programs have the ability to update themselves without exterior intervention, and that is precisely what happens when there is an EVOLP program rule of the form $assert(X) \leftarrow Y$. In the next time step, $X$ becomes a new rule of the program if $Y$ holds in the previous time step model. Since $d$ does not hold in the first model, the new rule $b \leftarrow not\ c$ is added to the program.

Also, since $e$ does not hold in the first model of the program, the new rule $assert(c \leftarrow not\ a)$ is added to it. In a nutshell, in the second time step the program becomes

$a \leftarrow not\ b.$

$b \leftarrow not\ c.$

$assert(b \leftarrow not\ c) \leftarrow not\ d.$

$assert(assert(c \leftarrow not\ a)) \leftarrow not\ e.$

$assert(c \leftarrow not\ a).$

and its new model is $\{b, assert(b \leftarrow not\ c), assert(assert(c \leftarrow not\ a)), assert(c \leftarrow not\ a)\}$.

A new evolution step occurs, due to the 'assert' atoms in the model, and the third time step program becomes

$a \leftarrow not\ b.$

$b \leftarrow not\ c.$

$c \leftarrow not\ a.$

$assert(b \leftarrow not\ c) \leftarrow not\ d.$

$assert(assert(c \leftarrow not\ a)) \leftarrow not\ e.$

$assert(c \leftarrow not\ a).$

Redundant effects of 'asserts' are discarded, i.e., asserting a rule that is already in the program results in keeping the rule in the program and not adding the duplicate one.

This is the time step where the problem arises. Under Stable Models semantics this EVOLP program no longer has a model due to the first three rules which constitute an Odd-Loop over Negation. This OLON was somehow camouflaged by the 'assert' EVOLP rules in the first time step program. As the natural evolution of the program unfolded it, the Odd-Loop Over Negation reached the 'surface' and revealed itself.

When one programs a knowledge-processing software agent with the ability to be updated (either by self-updates or by external ones) the need of an underlying semantics that copes with possible emergent Odd-Loops Over Negation cannot be simply refused. Otherwise, one incurs in the risk of seeing its agent breakdown due to a sudden lack of semantics.

Even if an agent does not perform self-updates, it can be externally updated. If the underlying semantics does not deal with Odd-Loops Over Negation, a malicious external agent can send our agent a new rule $a \leftarrow not\ a$. If our agent accepts this new rule and it runs on Stable Models semantics, it is doomed to lack of semantics forever.

Obviously, one can argument in favor of the beforehand detection and prevention of such OLONs. In the case of simple, direct Odd-Loops Over Negation — such as the case of $a \leftarrow not\ a$ — this is a simple process. But to detect a more elaborate OLON — like the one in our previous example 5.1.2 turns out to be a very computational expensive operation. In the worst case scenario, every rule of the program must be checked to ensure that there are no hidden OLONs. Moreover, every time a new rule is added to the knowledge base a full scan must be ran over the program to ensure it is OLON-free. This is a task that becomes more and more computationally expensive as the program gets bigger due to updates.

For all this, we believe the Revised Stable Models semantics can be a more effective and robust underlying semantics for EVOLP programs than classical Stable Models. And since Revised Stable Models is an extension to Stable Models each and every desirable property of the current standard 2-valued semantics is preserved.

## 5.2 REWERSE

REWERSE [37] is a research "Network of Excellence" (NoE) on "Reasoning on the Web" that is funded by the EU Commission and Switzerland within the "6th Framework Program" (FP6), Information Society Technologies (IST), Priority 2 under the project reference number 506779. REWERSE addresses the IST strategic action line "Semantic-based knowledge systems".

REWERSE's goals include the development of a coherent and complete, yet minimal, collection of inter-operable reasoning languages for advanced Web systems and applications. These languages, which should reach the level of open pre-standards amenable to submissions to standardization bodies such as the W3C [39], must comprehend mechanisms for the development of autonomous evolving and reacting Web resources (such as agents, knowledge bases, and others).

One of the Working Groups of REWERSE — namely Working Group I5 — is specially devoted to the definition of principles, languages, and semantics of evolving and reacting Web resources.

The existence of semantics for every Web resource, no matter what its course of evolution might be, is of upmost importance. Moreover, for usability and efficiency, one can easily understand that the ability to pose queries — which can be answered through a top-down approach procedure — is absolutely necessary. Nowadays, even the most simplistic databases allow users to pose simple queries. The user

expects the answering engine behind the interface to be sufficiently efficient to give him/her an answer in a short time period. Surely, in a knowledge-base with thousands of facts and rules, if a query simply depends on a few literals, the user expects the answer to come up quickly.

In a knowledge-base written in a rule-style declarative language, *a la* Prolog, this can only be possible if the underlying semantics is Relevant.

Moreover, to speed up computations of answers to queries, one expects to be possible to use tabling/memoizing techniques which make it possible to store intermediate results. Again, this is only possible if the underlying semantics is Cumulative.

In such a competitive and fast changing environment as the Internet, and more specifically the World Wide Web, the applications must meet the ever-growing demands of the users. The languages and pre-standards the REWERSE project is committed to develop are to be submitted to the Consortium and other organizations for approval and to become *de facto* standards. For the success of these approvals and the acceptance of the general community of computer scientists and professional software developers, it is necessary that the underlying reasoning mechanism of the Semantic Web guarantee a high level of efficiency. Relevancy and Cumulativity are two cornerstones for the building of an efficient reasoning engine, and we believe Revised Stable Models semantics can give a contribute here.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

Stable Models are the current state-of-the-art 2-valued semantics. Any new semantics that aspires to be considered as another useful underlying platform for 2-valued reasoning must be a natural extension of Stable Models (in the sense that every Stable Model of a NLP P is also a Revised Stable Model of the same NLP).

We propose this new semantics — Revised Stable Models — to the Knowledge Representation and Reasoning scientific community with the intent of giving a contribution to the diversification of available reasoning tools and mechanisms.

We propose Revised Stable Models semantics to be considered as another useful tool in the Knowledge Representation and Reasoning researchers' toolbox. A new tool that, by its convenient properties of Existence, Relevancy and Cumulativity, allows the implementation of handy top-down proof procedures, the use of tabling techniques, always guaranteeing the existence of a Model.

Since every Stable Model of a NLP is also a Revised Stable Model of the same NLP, the interested researcher can always check if the obtained Model is consistent, i.e., if it is also a Stable Model; or, otherwise, if there are some literals in the Model which were considered true by RAA reasoning.

The fact that Revised Stable Models semantics is an extension to the Stable Models semantics allowed us to conclude that the Stable Models semantics, under certain special conditions, also ensures the Existence of a Model, enjoys Relevancy and Cumulativity. These, as far as we were able to find, are new results about the SM semantics and may prove to be useful in opening new doors to practical implementations of SM-based systems.

## 6.2 Future Work

### 6.2.1 Extensions

The foundations of the Revised Stable Models semantics have been laid, and it them a new range of possibilities is open, a new form of resolving OLONs and ILSCs has been proposed. RSM semantics has been proposed for Normal Logic Programs and we have showed the useful properties it enjoys. The next step is to expand the Revised Stable Models semantics to Extended Logic Programs (with explicit negation) and to Generalized Logic Programs (with default negations in the heads of rules).

The first steps of future work taken in this area will consider these extensions of the RSM semantics, as well as its use for including the general RAA reasoning as a form of Belief Revision.

#### Extended Logic Programs and the Revised Answer-Sets

Extended Logic Program [2] take the knowledge representation formalism one step further than Normal Logic Programs by including Explicit Negation — sometimes referred to as Strong Negation. This allows the programmer to write rules like

$$\neg a \leftarrow b, \sim c$$

where $\neg$ corresponds to the Explicit Negation whereas $\sim$ corresponds to the Default Negation.

Under Extended Logic Programs (ELPs) one can, ultimately, have a program that, under some specific interpretation, derives an explicit contradiction.

*Example 6.2.1.* Extended Logic Program with explicit contradiction
Let $P$ be

$\neg a \leftarrow\sim b$

$a \leftarrow\sim b$

Since there is no rule for '$b$' in $P$, its unique Minimal Model is $\{a, \neg a\}$, which is explicitly contradictory.

ELPs, by its additional logic construct — the explicit negation — provide to the programmer a tool with a higher expressive power; at the cost of allowing explicit contradictions to arise. It is up to the programmer to ensure — if he/she wants to — the program is contradiction-free.

The version of the Stable Models semantics for Extended Logic Programs is generally known as *Answer-Set Programming* (ASP). The ASP semantics is *grosso modo* similar to the Stable Models semantics with the additional proviso that no Answer-Set (the equivalent to a Stable Model) can have an explicit contradiction, i.e., an atom and its explicit negation. The ASP semantics says that when in a ELP every Minimal Model has an explicit contradiction, then there is just one Answer-Set which is the whole Herbrand Base. Thus, the ASP semantics follows the *Ex Contradictio Quod Libet* principle.

Besides the issue of explicit contradiction, the ASP semantics, just like the Stable Models semantics, does not deal with OLONs nor with ILSCs. Keeping this in mind and following a similar procedure,

we believe an extension the Answer-Set Programming semantics can be built using the same principles behind Revised Stable Models.

A *Revised Answer Sets* (RAS) semantics, extending the Answer-Set Programming semantics, should not differ much from the RSM semantics and enjoy at least some of its properties.

### Generalized Logic Programs and the Revised GLP Semantics

Generalized Logic Programs (GLPs), similarly to what happens with ELPs, are an extension to Normal Logic Programs. In the case of GLPs there is no explicit negation, but the rules of the program are allowed to have default negated literals in the heads. Hence, the same potential problem of deriving a contradiction is present. The differences between ELPs and GLPs are subtle and they are not the main concern here.

An example of a GLP follows.

*Example 6.2.2.* Generalized Logic Program with contradiction
Let $P$ be

$\sim a \leftarrow\sim b$

$a \leftarrow\sim b$

Since there is no rule for '$b$' in $P$, its unique Minimal Model is $\{a, \sim a\}$, which is contradictory.

We believe that it is possible to define an extension to the Answer-Set Programming semantics used for GLPs such that this new *Revised Generalized Logic Programming* (RGLP) semantics takes the advantages of the Generalized Logic Programs expressive power, keeping at least some of the useful properties of the Revised Stable Models.

One would expect that the definition of the RGLP semantics should not differ much from the definition of the RAS semantics.

### Revised Well-Founded Semantics

An additional extension to the RSM semantics can be attained by entering the domain of 3-valued logics. This can be done by extending, with the already known mechanisms for dealing with OLONs and ILSCs, the Well-Founded Semantics.

Under the new Revised Well-Founded Semantics (RWFS) only the atoms in ELONs would have the *undefined* truth-value. Atoms involved in OLONs or in ILSCs would be treated in the same way as in the RSM semantics. Since the Well-Founded Semantics already guarantees the Existence of a model, is Relevant and Cumulative, the RWFS should keep these properties.

### Belief Revision

General RAA reasoning is the general procedure of Belief Revision which consists of finding one, or more, possible assumed hypothesis revision when a contradiction arises. In this way, a reduced form

of Belief Revision is embedded in Revised Stable Models through its inherent reduced form of RAA reasoning. Taking the RSM semantics one step further to Extended Logic Programs or to Generalized Logic Programs could yield the possibility of having a general Belief Revision mechanism imbedded in the semantics.

Having such kind of full Belief Revision imbedded in the semantics can free the top-level programmer from developing a complex Belief Revision system as it happens today. In this sense, a generalized form of Revised Stable Models, such as Revised Answer-Sets, can absorb the complexity of a general Belief Revision system.

Naturally, such an extended version of the RSM semantics will have a higher degree of complexity, since the burden of general Belief Revision will be imbedded in the semantics.

### 6.2.2 Further work

Besides investigating the possibilities of extending the Revised Stable Models semantics for ELPs, GLPs, and others, a formal comparison of the RSM semantics against other semantics (like Clark's Completion [13], Well-Founded Semantics [20] ), and other logics (Intuitionist Logic [27], Here-and-There Logic, and others) is one of the next main research steps to be taken. The Revised Stable Models semantics has just been born and it must be carefully studied and all its most important properties discovered and compared to other semantics' in order to fully understood. A thorough exploration of the properties of *Sustainable Sets* as presented in definition 2.5.1 is also in our research agenda.

Still on the theoretical side of further work, a formal and thorough complexity analysis of this semantics is also in our schedule. This is clearly a matter of major importance since it will ultimately give an insight on the computational usability of this semantics.

Also, the implementations described in this Thesis were developed with the intent of showing that they are feasible and usable. These implementations are very far from being efficient and optimized — that was not their current aim. Future work in Revised Stable Models includes the development of a more efficient top-down proof-procedure for solving queries *a la* Prolog, and also a more efficient RSM calculator — like the SModels [28]. A possible way of doing this efficient implementation is by means of a program transformation which guarantees that the Stable Models of the resulting transformed program are the Revised Stable Models of the original one. This way, after transforming the program, the resulting set of rules can be 'given' to a SModels implementation to obtain its Stable Models, i.e., the RSMs of the original program.

# Bibliography


[1] João Alcântara, Carlos Viegas Damásio, and Luís Moniz Pereira. Paraconsistent logic programs. In In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference of Logics In Artificial Intelligence, JELIA 2002*, volume 2424, pages 345–356. Springer, September 2002.

[2] José Júlio Alferes. *Semantics of Logic Programs with Explicit Negation*. PhD thesis, Universidade Noval de Lisboa, October 1993.

[3] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Procs. of the 8th European Conf. on Logics in Artificial Intelligence (JELIA'02)*, number 2424 in LNCS, pages 50–61. Springer, September 2002.

[4] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.

[5] José Júlio Alferes and Luís Moniz Pereira. *Reasoning with Logic Programming*, volume 1111 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[6] José Júlio Alferes, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. Lups - a language for udating logic programs. *Artificial Intelligence*, 138(1–2), 2002.

[7] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.

[8] Francisco Azevedo. *Constraint Solving over Multi-valued Logics - Application to Digital Circuits*, volume 91 of *Frontiers of Artificial Intelligence and Applications*. IOS Press, 2003.

[9] Federico Banti, José Júlio Alferes, and Antonio Brogi. Well founded semantics for logic program updates. In J. A. González C. Lemaître, C. A. Reyes, editor, *Advances in Artificial Intelligence - IBERAMIA 2004, 9th Ibero-American Conference on AI*, volume 3315 of *Lecture Notes in Computer Science*, pages 397–407. Springer-Verlag, 2004.

[10] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics - technical report cs-tr-2402. Technical report, University of Maryland, 1990.

[11] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics. *Journal of Automated Reasoning*, 8:345–366, 1992.

[12] Stefan Brass and Jürgen Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Program.*, 40(1):1–46, 1999.

[13] K. Clark. Negation as failure. In H.Gallaire and J.Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[14] Stefania Costantini. Contributions to the stable models semantics of logic programs with negation. *Theoretical Computer Science*, 149(2):231–255, 1995.

[15] Stefania Costantini. On the existence of stable models of non-stratified logic programs. *Under consideration for publication in Theory and Practice of Logic Programming*, 2003.

[16] Carlos Viegas Damásio and Luís Moniz Pereira. A paraconsistent semantics detecting contradiction support. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and NonMonotonic Reasoning, 4th Int. Conf.*, number 1265 in LNAI, pages 224–243. Springer, July 1997.

[17] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[18] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976. ISSN:0004-5411.

[19] François Fages. Consistency of clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.

[20] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[21] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *In Procs. of ICLP-88*, pages 1070–1080. International Conference on Logic Programming 88, 1988.

[22] João Alexandre Leite. *Evolving Knowledge Bases - Specification and Semantics*. IOS Press, 2003.

[23] João Alexandre Leite, José Júlio Alferes, Luís Moniz Pereira, H. Przymusinska, and T. C. Przymusinski. A language for multi-dimensional updates. *Electronic Notes in Theoretical Computer Science*, 70(5), 2002.

[24] Vladimir Lifschitz. Answer set planning. In *Proceedings of the International Conference on Logic Programming*, pages 23–37, 1999.

[25] Thomas Lukasiewicz. Probabilistic and truth-functional many-valued logic programming. Technical report, Institute für Informatik - Justus-Liebig-Universität Giessen, December 1998. IFIG Research Report 9809.

[26] Bamshad Mobasher, Don Pigozzi, and Giora Slutzki. Multi-valued logic programming semantics: An algebraic approach. *Theoretical Computer Science*, 171(1–2):77–109, 1997.

[27] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2004.

[28] Ilkka Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429, July 1997.

[29] Artificial Intelligence Centre of the Computer Science Department at Universidade Nova de Lisboa. Centria — artificial intelligence centre web site. Web Site.

[30] Luís Moniz Pereira. *Handbook of the Logic of Argument and Inference*, volume 1 of *Studies in Logic and Practical Reasoning*, chapter Philosophical Incidences of Logic Programming, pages 425–448. Elsevier Science, 2002.

[31] Luís Moniz Pereira and José Júlio Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on Artificial Intelligence*, pages 102–106. John Wiley & Sons, 1992.

[32] Luís Moniz Pereira and Alexandre Miguel Pinto. Revised stable models - a new semantics for logic programs. In *In Procs. Convegno Italiano di Logica Computazionale (CILC'04)*. Convegno Italiano di Logica Computazionale (CILC'04), July 2004. Parma, Italy.

[33] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence, a Sourcebook*, pages 321–367. North Holland, 1990.

[34] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–21, New York, NY, USA, 1989. ACM Press.

[35] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[36] Abhik Roychoudhury, K. Narayan Kumar, C.R.Ramakrishnan, and I.V. Ramakrishnan. A parametrized unfold/fold transformation framework for definite logic programs. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*. Springer-Verlag, 1999.

[37] The REWERSE Team. Rewerse - reasoning on the web with rules and semantics. WWW.

[38] WASP Team. Working group on answer set programming. WWW.

[39] W3C. World wide web consortium. WWW.

# Appendix A

# Proofs of Theorems

## A.1 Minimal and Classical Models Theorems, Lemmas and Corollaries

**Theorem A.1.1.** *Alternative Classical Model of P*

*If $M$ is a Minimal Classical Model of $P$, then $\Gamma_P^2(M)$ is a Classical Model of $P$.*

$$MM_P(M) \Rightarrow CM_P(\Gamma_P^2(M)) \tag{A.1}$$

*Proof.* Let us analyze in turn the cases where $RAA_P(M) = \emptyset$ and $RAA_P(M) \neq \emptyset$.

If $RAA_P(M) = \emptyset$ then $M = \Gamma_P(M)$, by definition of $RAA_P(M)$. In this case $M = \Gamma_P(M) = \Gamma_P^2(M)$; hence $M = \Gamma_P^2(M)$.

Since $M$ is a Minimal Classical Model of $P$, and $M = \Gamma_P^2(M)$, we conclude that $\Gamma_P^2(M)$ is also a Minimal Classical Model of $P$ and, necessarily, a Classical Model of $P$.

Let us now consider the case where $RAA_P(M) \neq \emptyset$.

Since $M$ is a Minimal Model of $P$, by definition of Minimal Model, any proper subset of $M$ is not a Classical Model of $P$. We know that $M \supseteq \Gamma_P(M)$ by [14]. Also, since $RAA_P(M) \neq \emptyset$ and $M = \Gamma_P(M) \cup RAA_P(M)$, we conclude that $\Gamma_P(M)$ is necessarily not a Classical Model of $P$.

By Theorem 2.3.1 we know that the rules of $P$ that are no longer satisfied in $P$ by $\Gamma_P(M)$ are those that were uniquely satisfied by the atoms that were in $M$ but are not in $\Gamma_P(M)$, i.e., the atoms in $RAA_P(M)$.

Let $U_P(M)$ — the 'unsupported rules' of P — denote the set of rules of P which are uniquely satisfied by the atoms in $RAA_P(M)$. Formally,

$$U_P(M) = \bigcup_{a \in RAA_P(M)} SUSR(P, a, M)$$

So, the rules in $P$ which are not satisfied by $\Gamma_P(M)$ are those in $U_P(M)$, so $CM_{P-U_P(M)}(\Gamma_P(M))$ holds.

In order to obtain another Classical Model of $P$, starting from $\Gamma_P(M)$ we need to add some more atoms such that the rules in $U_P(M)$ — those that are not satisfied in $\Gamma_P(M)$ — become now satisfied. The fact that these rules are not satisfied under $\Gamma_P(M)$ means that their heads are not in $\Gamma_P(M)$ and their bodies are satisfied by $\Gamma_P(M)$. I.e.,

$$\forall_{r \in U_P(M)} head(r) \notin \Gamma_P(M) \wedge \Gamma_P(M) \vdash_{\Gamma_P} body(r)$$

Since all the atoms in $\Gamma_P(M)$ are classically supported — otherwise they would not be in $\Gamma_P(M)$ — they will also be in $\Gamma_P(\Gamma_P(M))$. Moreover, all the heads of the rules in $U_P(M)$ will also be in $\Gamma_P(\Gamma_P(M))$ since their bodies are satisfied under $\Gamma_P(M)$. From this it follows that all rules in $P$ become satisfied under $\Gamma_P(\Gamma_P(M)) = \Gamma_P^2(M)$. Hence, $CM_P(\Gamma_P^2(M))$. □

**Corollary A.1.1.** $MM_P(M) \Rightarrow \Gamma_P^2(M) \supseteq \Gamma_P(M)$

*Proof.* It follows trivially from the previous theorem A.1.1. □

A generalization of the previous theorem can be made to Classical Models. We do that in the following

**Theorem A.1.2.** *The $\Gamma_P^2$ application is closed within the set of Classical Models of a Normal Logic Program*

*If $M$ is a Classical Model of $P$, then $\Gamma_P^2(M)$ is also a Classical Model of $P$.*

$$CM_P(M) \Rightarrow CM_P(\Gamma_P^2(M)) \tag{A.2}$$

*Proof.* By theorem A.1.1 we know that if $M$ is a Minimal Model of $P$ then $\Gamma_P^2(M)$ is a Classical Model of $P$.

We also know that the $\Gamma_P$ operator is anti-monotonic, i.e.,

$$M \supseteq M' \Rightarrow \Gamma_P(M) \subseteq \Gamma_P(M') \tag{A.3}$$

A Classical Model of a NLP is, by definition, a superset of some Minimal Model of $P$

$$CM_P(M') \Rightarrow \exists_{M \subseteq M'} MM_P(M)$$

Conversely, the other way around interpretation, says that if $M$ is a Minimal Model of $P$ then there is at least one superset of $M$ that is a Classical Model of $P$

$$MM_P(M) \Rightarrow \exists_{M' \supseteq M} CM_P(M') \tag{A.4}$$

So, let $M$ be a Minimal Model of $P$ and $M'$ be a Classical Model of $P$ such that $M' \supseteq M$. By equation A.3 we know that $\Gamma_P(M') \subseteq \Gamma_P(M)$, and again, $\Gamma_P(\Gamma_P(M')) \supseteq \Gamma_P(\Gamma_P(M))$, i.e., $\Gamma_P^2(M') \supseteq \Gamma_P^2(M)$.

Since we know $\Gamma_P^2(M)$ is a Classical Model of $P$, all the rules of $P$ are satisfied in $\Gamma_P^2(M)$. Also, since $\Gamma_P^2(M') \supseteq \Gamma_P^2(M)$ the only rules in $P$ that could no longer be satisfied are the ones that were *Satisfied by the Absence of a Literal* in $\Gamma_P^2(M)$. If the literal whose absence satisfied one of those rules appears in $\Gamma_P^2(M')$, so does the head of the same rule — the $\Gamma_P$ operator performs a transfinite iteration of $T_P$ obtaining all possible consequences (heads of rules whose body is true).

Hence, $\Gamma_P^2(M')$ also satisfies all rules in P, i.e., $\Gamma_P^2(M)$ is a Classical Model of $P$. □

**Lemma A.1.1.** *Existence of $\Gamma_P^{2k}$ fixed point for finite Classical Models*
*If $P$ is a NLP with a finite number $n$ of Classical Models then, there is at least one Classical Model of $P$, which is a fixed point of the $\Gamma_P^{2k}$ operator, for some $k \in \mathbb{N}$.*

$$\#CM(P) = n \wedge n \in \mathbb{N} \Rightarrow \exists_M CM_P(M) : \exists_{k\in\mathbb{N}} \Gamma_P^{2k}(M) = M$$

*Proof.* Let P be a NLP with a finite number of Classical Models, and let M be one (any) of the Classical Models of P.

By theorem A.1.2 we know that $\Gamma_P^2(M)$ is also a Classical Model of P. If $M = \Gamma_P^2(M)$, M is one of the fixed points of the $\Gamma_P^{2k}$ operator — where $k = 1$.

If $M \neq \Gamma_P^2(M)$ then we can apply again the $\Gamma_P^2$ operator — thus getting as a result $\Gamma_P^2(\Gamma_P^2(M)) = \Gamma_P^4(M) = \Gamma_P^{2k}(M)$, where $k = 2$. We can keep applying the $\Gamma_P^2$ operator until some $\Gamma_P^{2k}(M)$ equals some $\Gamma_P^{2j}(M)$, with $j < k$.

If after $n$ — the number of Classical Models of P — consecutive applications of the $\Gamma_P^2$ operator we have still not reached a fixed point (every previous application of $\Gamma_P^2$ gave as result a different Classical Model of P) then, inevitably, the $n + 1$th application of $\Gamma_P^2$ will get as a result a Classical Model which was already $\Gamma_P^{2j}(M)$ for some $0 \leq j \leq n$, since P has a finite number of Classical Models.

So, necessarily, $\exists_{0\leq j<\phi\leq n+1} \Gamma_P^{2\phi}(M) = \Gamma_P^{2j}(M)$

$\Gamma_P^{2j}(M)$ is thus a fixed point of the $\Gamma_P^{2\phi-2j} = \Gamma_P^{2(\phi-j)}$ operator. Since, by hypothesis, $j < \phi$, $k = \phi - j \geq 1$, $\Gamma_P^{2j}(M)$ is a fixed point of the $\Gamma_P^{2k}$ operator; where $k \geq 1 \Leftrightarrow k \in \mathbb{N}$. □

**Theorem A.1.3.** *If $M$ is a Classical Model then $\forall_{n\in\mathbb{N}} \Gamma_P^{2n}$ is also a Classical Model*

$$CM_P(M) \Rightarrow \forall_{n\in\mathbb{N}} CM_P(\Gamma_P^{2n}(M)) \tag{A.5}$$

*Proof.* The previous theorem A.1.2 says that, M being a Classical Model of P, $\Gamma_P^2(M)$ is also a Classical Model of P.

It it trivial to prove, by simple induction, that $CM_P(\Gamma_P^{2n}(M))$, if $CM_P(M)$, for any $n \in \mathbb{N}$. By theorem A.1.2 we know that

$$CM_P(M) \Rightarrow CM_P(\Gamma_P^2(M))$$

If $CM_P(\Gamma_P^{2(n-1)}(M))$ then, again by theorem A.1.2 we conclude

$$CM_P(\Gamma_P^2(\Gamma_P^{2(n-1)}(M))) \Leftrightarrow CM_P(\Gamma_P^{2(n-1)+2}(M)) \Leftrightarrow$$
$$CM_P(\Gamma_P^{2n-2+2}(M)) \Leftrightarrow CM_P(\Gamma_P^{2n}(M))$$

□

## A.2 Lemmas about the $\Gamma_P$ operator and Interpretations

**Lemma A.2.1.** *Existence of rule for classically supported atoms*
*Let I be an Interpretation for Normal Logic Program P. For every atom* `a` *in* $\Gamma_P(I)$ *there is at least one rule in P with head* `a`*, and that rule is satisfied only by* `a`*.*

$$a \in \Gamma_P(I) \Rightarrow \exists_{r \in P} head(r) = a$$

*Proof.* By definition of the $\Gamma_P$ operator

$$\Gamma_P(I) = lfp(T_P \uparrow^\omega (P/I))$$

where $lfp$ stands for *least fixed point* and the program division $P/M$ is the one defined in section 2.4.

The $\Gamma_P$ operator turns out to be the consecutive iteration of the $T_P$ operator until a fixed point is attained. Since the $T_P$ operator just collects heads of rules whose bodies are true in the interpretation, we know that $\Gamma_P(I)$ must be a subset of the heads of the rules of P, i.e., $\Gamma_P(I) \subseteq Heads(P)$. Moreover, $a \in \Gamma_P(I)$ iff there is at least one rule in $P$ with head $a$ and true body under $I$.

Formally,

$$a \in \Gamma_P(I) \Leftrightarrow \exists_{r \in P} head(r) = a \wedge I \vdash_{\Gamma_P} body(r)$$

Hence, if an atom `a` is in $\Gamma_P(I)$ then it must be the head of some rule of P, which concludes this proof. □

**Lemma A.2.2.** $a \in \Gamma_P(M) \Rightarrow \exists_{r \in P} head(r) = a \wedge r \in SUSR(P, a, M)$

*Proof.* By lemma A.2.1 we know that $a \in \Gamma_P(M) \Rightarrow \exists_{r \in P} head(r) = a \wedge M \vdash_{\Gamma_P} body(r)$

Since, under $M$, the whole body of the rule is true, the only way the rule can be satisfied is by having the atom in its head also in the interpretation. So, for every rule $r$ of $P$ such that $head(r) = a \wedge M \vdash_{\Gamma_P} body(r)$ holds, we conclude that $r$ is in the *Set of Uniquely Satisfied Rules* of P, by literal $a$ — the head of the rule.

$$head(r) = a \wedge M \vdash_{\Gamma_P} body(r) \Rightarrow r \in SUSR(P, a, M)$$

By lemma A.2.1 it follows that

$$a \in \Gamma_P(M) \Rightarrow \exists_{r \in P} head(r) = a \wedge r \in SUSR(P, a, M)$$

□

## A.3 Revised Stable Models Theorems and Lemmas

**Lemma A.3.1.** *Existence of rule for atoms in a Revised Stable Model*
*If M is a Revised Stable Model of a NLP P then, for every atom a in M there is at least one rule in P with head a.*

$$RSM_P(M) \Rightarrow \forall_{a \in M} \exists_{r \in P} head(r) = a$$

*Proof.* By definition of Revised Stable Model,

$$RSM_P(M) \wedge a \in M \Rightarrow (a \in \Gamma_P(M) \vee a \in RAA_P(M))$$

If $a \in \Gamma_P(M)$ then, by lemma A.2.1 we know that $\exists_{r \in P} head(r) = a$.

If $a \in RAA_P(M)$, by definition of RSM we know that $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$. This means that $\exists_{\alpha \geq 2} a \in \Gamma_P^\alpha(M)$.

We know that $\Gamma_P^\alpha(M)$ is just a shorthand notation for $\Gamma_P(\Gamma_P^{\alpha-1}(M)$, and we also know that $\Gamma_P^{\alpha-1}(M)$ is just an interpretation of P. Let us call this interpretation $I = \Gamma_P^{\alpha-1}(M)$. Then, we have that $\exists_I a \in \Gamma_P(I)$.

From lemma A.2.1 we conclude again that $\exists_{r \in P} head(r) = a$. □

**Theorem A.3.1.** *Stable Models Extension*
*Every Stable Model of a NLP P is a Revised Stable Model of P — Theorem 4.4.1*
*Formally,*

$$\forall_M SM_P(M) \Rightarrow RSM_P(M)$$

*Proof.* Let $P$ be a Normal Logic Program, and let $M$ be such that $SM_P(M)$ holds, i.e., $M$ is a Stable Model of $P$.

Every Stable Model $M$ of a NLP $P$ is a Minimal Classical Model of $P$; thus $M$ complies with the first condition of the RSM definition.

By definition of Stable Model, $M = \Gamma_P(M)$, and so $RAA_P(M) = M - \Gamma_P(M) = \emptyset$.

Since $RAA_P(M) = \emptyset$, the second condition of the RSM definition is trivially satisfied

$$\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M) = \emptyset$$

Also, since $RAA_P(M) = \emptyset$ it trivially complies with the third condition of the RSM definition.

$M$ complying with all three conditions of the RSM definition, it is proven that $M$ is a Revised Stable Model of $P$. □

**Theorem A.3.2.** *Existence*
*Every Normal Logic Program has, at least, one Revised Stable Model*

*Proof.* We have already showed, in Theorem 4.4.1 that every Stable Model of a NLP $P$ is also a Revised Stable Model of $P$. So, the proof of the Existence of Revised Stable Models for any NLP $P$ is reduced to the case where $P$ has no Stable Models at all — otherwise the Existence of Revised Stable Models would be trivially assured. Hence, we now concentrate only on the case where $P$, a Normal Logic Program, has no Stable Models.

Every NLP $P$ has a Well-Founded Model [20], so it is always possible to calculate $WFM(P) =< WFM^+(P), WFM^-(P), WFM^u(P) >$, where $WFM^+(P)$ is the subset of *positive* atoms of $WFM(P)$, $WFM^-(P)$ is the subset of *negative* atoms of $WFM(P)$, and $WFM^u(P)$ is the subset of *undefined* atoms of $WFM(P)$.

Likewise, it is always possible to calculate the *Residual Program of* $P$ — $P^R$, — i.e., the most simplified version of $P$ which preserves its semantics. To calculate $P^R$ follow these steps:

1. calculate the $WFM(P)$
2. delete from $P$ all the rules with $\sim a$ in the body where $a \in WFM^+(P)$
3. delete from the bodies of rules all the occurrences of $a$ where $a \in WFM^+(P)$
4. delete from $P$ all the rules with head $a$ where $a \in WFM^+(P)$
5. add all the atoms of $WFM^+(P)$ to $P$ as facts
6. delete from the bodies of rules all the occurrences of $\sim b$ where $b \in WFM^-(P)$
7. delete from $P$ all the rules with $b$ in the body where $b \in WFM^-(P)$
8. delete from $P$ all the rules with head $b$ where $b \in WFM^-(P)$

We name the resulting program $P^R$ the *Residual Program of* $P$. By construction, $P^R$ has the same semantics as $P$.

Since every NLP has at least one Minimal Model we can calculate one such Minimal Model $M$ of $P^R$. We can also always calculate its $\Gamma$ consequences, i.e., $\Gamma_{P^R}(M)$. Again, we can make a further simplification of $P^R$ by doing with $\Gamma_{P^R}(M)$ the steps 2–5 of the previous Residual Program calculation procedure: deleting from $P^R$ all the rules that contradict $\Gamma_{P^R}(M)$ and adding $\Gamma_{P^R}(M)$ as facts. We will name $P^{R \times \Gamma_{P^R}(M)}$ to the resulting program. By construction, it is always possible to build such a program. Clearly, all the Minimal Models of $P^{R \times \Gamma_{P^R}(M)}$ are also Minimal Models of $P$.

Now we add to $P^{R\times\Gamma_{PR}(M)}$ all the *top literals of all active OLONs and all active ILSCs under M* as facts. And now we reiterate this whole process (starting from the Residual Program calculation) at most $\omega$ times — where $\omega$ is the first limit ordinal — until a fixed-point program is achieved. By construction, the Minimal Models of the resulting program are Minimal Models of $P$. Also by construction, all Minimal Models of the resulting program respect the third condition of the RSM definition

$$RAA_P(M) \text{ is sustainable}$$

By construction, in each step of the transformation process we delete some rules and add a few facts to the program; this ensures that all Minimal Models of the subsequent intermediate programs of the transformation process include all added facts — thus "respecting" (accordingly to the notion of "respect" we defined in 2.5.1) those added facts.

Doing this process with every Minimal Model $M_i$ of $P$ — where $\{M_1, M_2, \ldots, M_m\}$ is the set of Minimal Models of $P$ — we will obtain, at most, $n$ resulting fixed-point programs $P^i$. By construction, every Minimal Model of every $P^i$ is a Minimal Model of $P$ and it respects the third condition of the RSM definition.

Let $M_P$ denote the set of Minimal Models of all $P^i$, i.e.,

$$M_P = \{M_j : \forall_{j\leq n} MM_{P^i}(M_j)\}$$

In subsection 2.4.4 we learned that only NLPs with OLONs or ILSCs with an infinite number of default negated *atoms involved in the ILSC* can have no Stable Models. Since this is the case we are studying now, the NLP $P$ we are considering must have at least one OLON or at least one such ILSC (or both, in any number).

Let us now consider $M \in M_P$. From propositions 2.4.2 and 2.4.5 we know that for every *top literal* $\lambda_i$ of any *active OLON under M*, $\lambda_i \in RAA_P(M) \wedge \Gamma_P^{n+1} \geq \{\lambda_i\}$, where $n$ is the length of the OLON. If $P$ has $m$ *active OLONs* under $M$ with $n_1, n_2, \ldots, n_m$ as respective lengths, then, with at most $\alpha = (n_1+1)(n_2+1)\ldots(n_m+1)$, we know that $\Gamma_P^\alpha(M) \supseteq \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$, where $\lambda_1, \lambda_2, \ldots, \lambda_m$ are the *top literals of the active OLONs under M*.

So, for all $\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ such that each $\lambda_i$ is the *top literal of an active OLON under M*, it is guaranteed that $\exists_\alpha \Gamma_P^\alpha(M) \supseteq \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$.

Since the minimum length of an OLON is 1, and by proposition 2.4.5 we know that $\Gamma_P^{n+1} \geq \{\lambda_i\}$ we conclude that $\alpha \geq 2$ must hold. Hence, finally we conclude that

$$\exists_{\alpha\geq 2} \Gamma_P^\alpha(M) \supseteq \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$$

Consider now, according to [19], the remaining case: the $\leq_+$ relation in $P$ is non-well-founded because there is an infinitely long decreasing chain along $\leq_+$, i.e., there is at least one ILSC $C$ with an infinite number of default negations, and $C$ *is active under M*.

Let $C$ be one of such ILSC in $P$ with $\{l_1, l_2, \ldots, l_n, \ldots\}$ as the *atoms involved in the ILSC*.

If the ILSC $C$ in $P$ with the infinite number of negations is the only ILSC, then, apart from possible OLONs in $P$, there are two Stable Models in $P$: $I_1$ which corresponds to the set of *evenly negated* atoms and their consequences, and $I_2$ which corresponds to the set of *oddly negated* atoms and their consequences. It is easy to verify that if $M \supseteq I_1$ then $I_1 \subseteq \Gamma_P(M)$, and also if $M \supseteq I_2$ then $I_2 \subseteq \Gamma_P(M)$.

Let us formalize this intuitive idea and see why it must hold.

Consider Model $I_1 = \{a : l \leq_+ a\} \cup \{l\}$, such that $l$ is the *top atom of the ILSC* $C$, and $C$ has an infinite number of rules with at least one default negated atom *involved in the ILSC* in the body, as just informally described.

Since there are no loops in the ILSC, if $M \supseteq I_1$, in applying the $\Gamma_P$ operator — described in 2.4 — with $M$, the program division part will eliminate all the rules $r \in P$ such that $\exists_{l_{i+1} \in M} \sim l_{i+1} \in body(r)$. By definition of ILSC, we know that rule $r$ is such that $head(r) = l_i$. Since we assumed $I_1$ to be $\{a : l \leq_+ a\} \cup \{l\}$, i.e., the set of *atoms involved in the ILSC* which have an even number of default negations in the dependency graph from the *top literal of the ILSC* to themselves, and that the ILSC is *active under* $M$, we must conclude that $l_i$ — the head of the rule with $\sim l_{i+1}$ in the body — is not in $\Gamma_P(M)$.

Consequently, since the ILSC is *active under* $M$, $\sim l_i$ will be true in $\Gamma_P(M)$. Hence, $\forall_{r' \in C} \sim l_i \in body(r') \Rightarrow M \vdash body(r')$, where $C$ is the ILSC, and so $l_{i-1} = head(r') \in \Gamma_P(M)$.

Moreover, since $l_{i-1} \in \Gamma_P(M)$ and $l_{i-1} = head(r') \wedge \sim l_i \in body(r')$ and $head(r) = l_i \wedge \sim l_{i+1} \in body(r)$ and $l_{i+1} \in I_1$, we must conclude that $l_{i-1} \in I_1$ also. Hence, all the atoms in $I_1$ are in $\Gamma_P(M)$, i.e., $M \supseteq I_1 \Rightarrow I_1 \subseteq \Gamma_P(M)$ and none of the atoms in $I_1$ is in $RAA_P(M)$. An analogous reasoning can be used for $I_2$.

This settled, since we assumed that at least one of the *atoms involved in the ILSC* is in $RAA_P(M)$, we are forced to conclude that the ILSC with an infinite number of rules with default negated atoms in the bodies is not the cause of those RAA-supported atoms.

So, since the atoms in $I_1$ ($I_2$), for instance, are in $\Gamma_P(M)$, the RAA atoms that are *involved in an ILSC* must not be in $I_1$ ($I_2$). Considering the case of $I_1$, the RAA-supported atoms *involved in the ILSC* must thus be the ones in $I_2$ (and vice-versa). This means that both $I_1$ and $I_2$ must be subsets of $M$. In this case, all the rules in the ILSC with default negated atoms (which are *involved in the ILSC*) in the bodies will have their bodies false under $M$. Hence, none of the $I_1$ nor $I_2$ atoms will be in $\Gamma_P(M)$. On the other hand, since $\Gamma_P(M)$ has none of the $I_1$ and $I_2$ atoms, $\Gamma_P(\Gamma_P(M))$ will have all of them — since under $\Gamma_P(M)$ all the $\sim l_i$ are true.

So we conclude that $\Gamma_P^2(M) \supseteq I$, where $I \subseteq RAA_P(M)$ is the set of *atoms involved in the ILSC*.

If any one of the $l_i$ atoms is in $RAA_P(M)$ then we know that $\forall_{r \in P} head(r) = l_i \Rightarrow M \nvdash_{\Gamma_P} body(r)$; and either $M \nvdash body(r)$ or $M \vdash body(r)$.

In case $M \nvdash body(r)$ we know that the rule $r$ is *satisfied by the body* and so $r \notin SUSR(P, l_i, M)$. In case $M \vdash body(r)$ the only possibility is that $\exists_{b \in body(r)} b \in M \wedge b \notin \Gamma_P(M)$, i.e., $\exists_{b \in body(r)} b \in RAA_P(M)$.

So, if $r \in SUSR(P, l_i, M)$ then, by definition of $ILSC$, either $head(r) = l_{i-1} \wedge \sim l_i \in body(r)$ or $\exists_{b \in RAA_P(M)} b \in body(r)$.

Let us consider the case where $head(r) = l_{i-1} \wedge \sim l_i \in body(r) \wedge \nexists_{b \neq l_i \wedge b \neq l_{i-1} \in body(r)} b \in RAA_P(M)$. We know that $M - \{l_i\} \vdash_{\Gamma_P} body(r)$. If we take $M - L$, where $L = \{l_i : l_i \in RAA_P(M) \wedge (\exists_{r \in ILCS} head(r) = l_{i-1} \wedge \sim l_i \in body(r)) \wedge \nexists_{b \neq l_i \wedge b \neq l_{i-1} \in body(r)} b \in RAA_P(M)\}$, then $M - L \vdash_{\Gamma_P} body(r)$. Now, the only atoms in $\{l_1, l_2, \ldots, l_n, \ldots\} - L$ which are in $RAA_P(M)$ are those $\{l_j, l_k, \ldots, l_x, \ldots\}$ where $\exists_{b \in RAA_P(M)} b \in body(r_j) \wedge head(r_j) = l_j$ holds, i.e., they are the $RAA_P(M)$ atoms which are in the $RAA_P(M)$ set because they depend on other $RAA_P(M)$ atoms.

Thus, if we take $M - RAA_P(M) = \Gamma_P(M)$ and calculate its $\Gamma_P$ consequences we will get all the $l_i$ atoms which are heads of rules of the ILSC. The $\Gamma_P$ consequences of $M - RAA_P(M)$ are simply $\Gamma_P(M - RAA_P(M)) = \Gamma_P(\Gamma_P(M)) = \Gamma_P^2(M)$.

Thus, considering only the $RAA_P(M)$ atoms which are involved in an ILSC — let us name it $I$, — we conclude that $\Gamma_P^2(M) \supseteq I$.

Also, since the $\Gamma_P$ operator performs a transfinite iteration of the $T_P$ operator, we will also get all the other atoms of the $RAA_P(M)$ set that depend on other $RAA_P(M)$ atoms — be they involved in an ILSC or *top literals of OLONs*.

We know that $RAA_P(M)$ atoms are either *top literals of active OLONs* or *atoms involved in ILSCs*. We already know that if $RAA_{OLON} \subseteq RAA_P(M)$ is such that every atom $a \in RAA_{OLON}$ is a *top literal of an active OLON under* $M$, then

$$\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_{OLON}$$

Also, we just concluded that, if $RAA_{ILSC} \subseteq RAA_P(M)$ is the subset of *atoms involved in an active ILSC under* $M$, then

$$\Gamma_P^2(M) \supseteq RAA_{ILSC}$$

Since we assumed that $P$ has no Stable Models, we know that every RAA set is partitioned into *top literals of active OLONs*, atoms that depend positively on such *top literals* — which together make up $RAA_{OLON}$, — and *atoms involved in active ILSCs* and atoms that depend positively on such ILSC atoms — which is $RAA_{ILSC}$, — we conclude that, at most,

$$\Gamma_P^{2\alpha}(M) \supseteq RAA_P(M)$$

where $\alpha$ is such that $\Gamma_P^\alpha(M) \supseteq RAA_{OLON}$.

Hence, finally, we proved that

$$\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$$

which concludes our proof of Existence of at least one RSM for every NLP.

**Lemma A.3.2.** *Existence of Supermodel*
*Let $P$ be a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset$, $P_a = Rel(P, a)$, and $M_a$ a RSM of $P_a$.*

*There is at least one RSM $M$ of $P$ such that $M \supseteq M_a$.*

$$\forall_{M_a} RSM_{P_a}(M_a) \Rightarrow \exists_{RSM_P(M)} M \supseteq M_a$$

*Proof.* Let $P$ be a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset$, $P_a = Rel(P, a)$, and $M_a$ a RSM of $P_a$.

By theorem 4.4.2 we know that every NLP has at least one RSM. So, let $M$ be a RSM of $P_{\overline{a}} \cup M_a$. Clearly $M \supseteq M_a$, since all the atoms in $M_a$ are facts in $P_{\overline{a}} \cup M_a$.

Since $M_a$ is a RSM of $P_a$ and $M$ is a RSM of $P_{\overline{a}} \cup M_a$, it follows trivially that $M$ is a Minimal Model of $P_a \cup P_{\overline{a}} = P$.

Since $M_a$ is a RSM of $P_a$ we know that $\exists_{\alpha \geq 2} \Gamma^{\alpha}_{P_a}(M_a) \supseteq RAA_{P_a}(M_a)$. Moreover, since $M$ is a RSM of $P_{\overline{a}} \cup M_a$ we also know that $\exists_{\alpha' \geq 2} \Gamma^{\alpha'}_{P_{\overline{a}} \cup M_a}(M) \supseteq RAA_{P_{\overline{a}} \cup M_a}(M)$. Hence, $\exists_{2 \leq \alpha'' \leq \alpha\alpha'} \Gamma^{\alpha''}_{P}(M) \supseteq RAA_P(M)$.

Again, since $M_a$ is a RSM of $P_a$, we know that $RAA_{P_a}(M_a)$ is sustainable. Also, $RAA_{P_{\overline{a}} \cup M_a}(M)$ is sustainable. We know that $RAA_P(M) = M - \Gamma_P(M) = M - \Gamma_{P_{\overline{a}} \cup \Gamma_{P_a}(M_a)}(M)$. Also, $RAA_{P_{\overline{a}} \cup M_a}(M) = M - \Gamma_{P_{\overline{a}} \cup M_a}(M) = M - \Gamma_{P_{\overline{a}} \cup \Gamma_{P_a}(M_a) \cup RAA_{P_a}(M_a)}(M)$. Naturally, $RAA_P(M) \supseteq RAA_{P_{\overline{a}} \cup M_a}(M)$, and the atoms in $RAA_P(M) - RAA_{P_{\overline{a}} \cup M_a}(M)$ are just the $RAA_{P_a}(M_a)$ plus the atoms of $RAA_P(M)$ which depend positively on atoms of $RAA_{P_a}(M_a)$. We already know that both $RAA_{P_a}(M_a)$ and $RAA_{P_{\overline{a}} \cup M_a}(M)$ are sustainable. The only possibility for $RAA_P(M)$ to be not sustainable is, by definition of sustainable set, to have some atom $a$ such that $RAA_P(M) - \{a\}$ is sustainable and $a \notin \Gamma_{P \cup (RAA_P(M) - \{a\})}(WFM(P \cup (RAA_P(M) - \{a\})))$. We have just seen that the atoms in $RAA_P(M)$ which are not in $RAA_{P_{\overline{a}} \cup M_a}(M)$ are $RAA_{P_a}(M_a)$ and the ones that depend positively on them. So, as soon as we add some atom of $RAA_{P_a}(M_a)$ to $P$ that atom becomes *true* in the $WFM$; and this can only render all other atoms which depend positively on that atom as *true* or else they remain *undefined*. Thus, there is no possibility $RAA_P(M)$ is not sustainable.

$M$ complying with all three RSM definition conditions, since $M$ is such that $RSM_{P_{\overline{a}} \cup M_a}(M)$ holds, then $M \supseteq M_a \wedge RSM_P(M)$ also holds. □

**Lemma A.3.3.** *Decomposition of the RAA set*
*Let $P$ be a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset$, and $P_a = Rel(P, a)$, and $M$ a RSM of $P$.*

$$RAA_P(M) = RAA_{P_a}(M_a) \cup \left(RAA_{P_{\overline{a}} \cup \Gamma_{P_a}(M_a)}(M) - M_a\right)$$

*Proof.* Let $P$ be a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset$, and $P_a = Rel(P, a)$, and $M$ a RSM of $P$.

From lemma A.3.1 we know that $M \subseteq Heads(P)$. So, it is possible to partition $M$ in the following way: $M = M_a \cup M_{\overline{a}}$, where $M_a \subseteq Heads(P_a)$ and $M_{\overline{a}} \subseteq Heads(P_{\overline{a}})$.

Since $RAA_P(M) \subseteq M$, we can also partition $RAA_P(M)$ into $RAA_{P_a}$ and $RAA_{P_{\overline{a}}}$, such that $RAA_{P_a} \subseteq M_a$ and $RAA_{P_{\overline{a}}} \subseteq M_{\overline{a}}$.

Naturally, $RAA_{P_a} \subseteq Heads(P_a)$ and $RAA_{P_{\overline{a}}} \subseteq Head(P_{\overline{a}})$.

Since $P_a = Rel(P, a)$, none of the rules in $P_{\overline{a}}$ is used by the $\Gamma$ operator to obtain as conclusion any head of $P_a$. By $RAA_{P_a} \subseteq M_a$ we know that $RAA_{P_a} = RAA_{P_a}(M_a)$. Hence, $RAA_{P_{\overline{a}}}(M) = RAA_P(M) - RAA_{P_a}(M_a) = (M - \Gamma_P(M)) - (M_a - \Gamma_{P_a}(M_a)) = M - \Gamma_P(M) - M_a$, since $\Gamma_{P_a}(M_a) \subseteq \Gamma_P(M)$.

Since $P \supseteq P_a$ and $P_a = Rel(P, a)$, we know that $\Gamma_{P_a}(M_a) = \Gamma_{P_a}(M)$, because none of the atoms in $M - M_a$ can be used by the $\Gamma$ operator to derive anything, or prevent anything from being derived, under $P_a$.

On the other hand, the atoms in $M_a$ can be used by the $T_P$ operator iteration performed by the $\Gamma$ operator to derive atoms under $P - P_a = P_{\overline{a}}$. So, $\Gamma_P(M) = \Gamma_{P_a}(M_a) \cup \Gamma_{P_{\overline{a}\cup\Gamma_{P_a}(M_a)}}(M)$.

Now, $RAA_P(M) - RAA_{P_a}(M_a) = M - \Gamma_P(M) - M_a = M - (\Gamma_{P_a}(M_a) \cup \Gamma_{P_{\overline{a}\cup\Gamma_{P_a}(M_a)}}(M)) - M_a = RAA_{P_{\overline{a}\cup\Gamma_{P_a}(M_a)}}(M) - M_a$. Hence $RAA_P(M) = RAA_{P_a}(M_a) \cup \big(RAA_{P_{\overline{a}\cup\Gamma_{P_a}(M_a)}}(M) - M_a\big)$. □

**Lemma A.3.4.** *Existence of submodel*
*Let $P$ be a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset$, and $P_a = Rel(P, a)$.*

*For every $M$ that is a Revised Stable Model of $P$ there is at least one $M_a$, subset of $M$, such that $M_a$ is a Revised Stable Model of $Rel(P, a) = P_a$.*

$$\forall_M RSM_P(M) \Rightarrow \exists_{M_a} RSM_{P_a}(M_a) \wedge M \supseteq M_a \tag{A.6}$$

*Proof.* Let $M$ be a RSM of $P$, $RSM_P(M)$. We know that $Rel(P, a) = P_a \subseteq P$. Since $M$ is a RSM of $P$ we know it is a Minimal Model of $P$ — by definition of Minimal Model. This means that all the rules in $P$ are minimally satisfied by $M$.

Let $M_a = M - Heads(P_{\overline{a}})$. $M_a$ is then the subset of $M$ which is a Minimal Model of $P_a$.

If $M_a = \Gamma_{P_a}(M_a)$, i.e., $M_a$ is a Stable Model of $P_a$, and by Theorem 4.4.1 we already know that $M_a$ is a RSM of $P_a$.

Let us now analyze the remaining case where $M_a$ is not a Stable Model of $P_a$.

Since $M \subseteq Heads(P)$ and $M_a = M - Heads(P_{\overline{a}})$, we conclude that $M_a \subseteq Heads(P_a)$, and also $RAA_{P_a}(M_a) \subseteq RAA_P(M)$.

Knowing $M$ is a RSM of $P$, we know that $\exists_{\alpha\geq 2}\Gamma_P^\alpha(M) \supseteq RAA_P(M) \supseteq RAA_{P_a}(M_a)$.

So, clearly, $\exists_{\alpha'\geq 2}\Gamma_{P_a}^{\alpha'}(M_a) \supseteq RAA_{P_a}(M_a)$, and $\alpha' \leq \alpha$ such that $\exists_{\alpha\geq 2}\Gamma_P^\alpha(M) \supseteq RAA_P(M)$.

Also, since $M$ is a RSM of $P$ we know that $RAA_P(M)$ is *sustainable*. By lemma A.3.3 we know that $RAA_P(M) = RAA_{P_a}(M_a) \cup \big(RAA_{P_{\overline{a}\cup\Gamma_{P_a}(M_a)}}(M) - M_a\big)$. This means that $RAA_{P_a}(M_a) \subseteq RAA_P(M)$. $RAA_{P_a}(M_a) \subseteq M_a$, and by the recursive nature of the *Sustainability* definition, since $M_a$ is a Minimal Model of $P_a$ and $RAA_P(M)$ is *sustainable* we conclude that $RAA_{P_a}(M_a)$ is also *sustainable*.

$M_a$ complying with all three RSM definition conditions we conclude that $M_a \subseteq M$ is a RSM of $P_a = Rel(P, a) \subseteq P$, where $RSM_P(M)$. □

**Theorem A.3.3.** *The Revised Stable Models semantics is Relevant (Theorem 4.4.3)*

*If $P$ is a NLP such that $P = P_a \cup P_{\overline{a}} \wedge P_a \cap P_{\overline{a}} = \emptyset \wedge P_a = Rel(P, a)$, then*

$$(\forall_M RSM_P(M) \Rightarrow a \in M) \Leftrightarrow (\forall_{M_a} RSM_{P_a}(M_a) \Rightarrow a \in M_a)$$

*Proof.* If we assume by hypothesis that $\forall_{M_a} RSM_{P_a}(M_a) \Rightarrow a \in M_a$ then, by lemma A.3.4 it is trivial to conclude that $\forall_M RSM_P(M) \Rightarrow a \in M$.

On the other hand, let us now consider that $\forall_M RSM_P(M) \Rightarrow a \in M$. By definition 2.2.8 we know that $(P = P_a \cup P_{\overline{a}}) \wedge (P_{\overline{a}} \cap P_a = \emptyset)$, where $P_a$ stands for $Rel(P, a)$.

We are assuming that $\forall_M RSM_P(M) \Rightarrow a \in M$. Since $a \in M$ then either $a \in \Gamma_P(M)$ or $a \in RAA_P(M)$. In this second case — where $a \in RAA_P(M)$ — by definition of RSM, $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M) \ni a$. We can thus trivially conclude that

$$a \in M \Rightarrow \exists_{n \in \mathbb{N}} a \in \Gamma_P^n(M)$$

By lemma A.2.2 we know that $a \in \Gamma_P(I) \Rightarrow \exists_{r \in P} head(r) = a \wedge r \in SUSR(P, a, I)$. Clearly, such a rule $r$ with head $\mathtt{a}$ is in $Rel(P, a) = P_a$. Since $r \in P_a \wedge head(r) = a \wedge a \in \Gamma_P(M)$ then we conclude that $a \in \Gamma_{P_a}(M)$.

By lemma A.3.2 we know that $\forall_{M_a} RSM_{P_a}(M_a) \Rightarrow \exists_{M \supseteq M_a} RSM_P(M)$. Since $a \in \Gamma_{P_a}(M)$ it follows trivially that $a \in \Gamma_{P_a}(M_a)$, and so $a \in M_a$. From this we conclude that $\forall_{M_a} RSM_{P_a}(M_a) \Rightarrow a \in M_a$. □

**Theorem A.3.4.** *The Revised Stable Models semantics is Cumulative (Theorem 4.4.4)*

$$\forall_P NLP(P) \Rightarrow \big(\forall_{a,b} a \in RSMS(P) \wedge b \in RSMS(P) \Rightarrow b \in RSMS(P \cup \{a\})\big)$$

*Proof.* Let $P$ be a NLP and $a$ and $b$ be such that $a \in RSMS(P) \wedge b \in RSMS(P)$. Let us also assume that $b \notin RSMS(P \cup \{a\})$. In this case we know that $\exists_M RSM_{P \cup \{a\}}(M) \wedge b \notin M$.

Since we know that $\forall_{M'} RSM_P(M') \Rightarrow b \in M'$, by corollary 2.3.1 $\forall_{M'} RSM_P(M') \Rightarrow \exists_{r \in P} r \in SUSR(P, b, M')$. If $\exists_M RSM_{P \cup \{a\}}(M) \wedge b \notin M$ then $\forall_{M'} RSM_P(M') \Rightarrow \forall_{r \in SUSR(P,b,M')} r \notin SUSR(P \cup \{a\}, b, M)$ — in fact, $SUSR(P \cup \{a\}, b, M) = \emptyset$.

Since $r \in SUSR(P, b, M')$ we know that either $head(r) = b \wedge M' \vdash body(r)$ or $\sim b \in body(r) \wedge M' \vdash body(r) - \{\sim b\}$.

In the first case — $head(r) = b \wedge M' \vdash body(r)$ — the only way the addition of $\mathtt{a}$ as a fact to $P$ can make $r \notin SUSR(P \cup \{a\}, b, M)$ hold is if $\sim a \in body(r)$. In that case, since we assumed $\forall_{M'} RSM_P(M') \Rightarrow a \in M'$, we must conclude that $r \notin SUSR(P, b, M')$, because $\mathtt{a}$ also satisfies the rule $r$. This contradicts the previously assumed hypothesis that $r \in SUSR(P, b, M')$.

The only possibility left is $\sim b \in body(r) \wedge M' \vdash body(r) - \{\sim b\}$. By the same kind of reasoning, $\sim a$ must not be in $body(r)$, otherwise since $\forall_{M'} RSM_P(M') \Rightarrow a \in M'$, we conclude $r \notin SUSR(P, b, M')$. So the only possibility is that $head(r) = a$; but also this must not be the case, otherwise $S(P, r, a)$, and $r \notin SUSR(P, b, M')$. This again contradicts the initial hypothesis that $r \in SUSR(P, b, M')$.

Hence, we must conclude that $\exists_M RSM_{P \cup \{a\}}(M) \wedge b \notin M$ does not hold, i.e., the Revised Stable Models semantics is Cumulative. □

# Appendix B

# Examples

In this chapter we present a series of examples we used throughout the process of definition of the Revised Stable Models semantics. These NLPs, though simple, test very precise conditions and situations and revealed themselves as a valuable tool for testing the conditions of the definition against the intended result.

For simplicity and for the sake of minimality of space used, in the following examples we only show the Minimal Models of the NLPs instead of all its possible interpretations.

## B.1 Examples vs RSM conditions

*Example B.1.1.* Simple Odd-Loop

$a \leftarrow\sim a$

| Candidate Min.Mod. M | $\{a\}$ |
|---|---|
| $\Gamma_P(M)$ | $\emptyset$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{a\}$ |
| $\exists_{\alpha \geq 2} \Gamma_P^{\alpha}(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes |
| Is RSM ? | Yes |

*Example B.1.2.* Simple Even-Loop

$a \leftarrow\sim b$

$b \leftarrow\sim a$

| Candidate Min.Mod. M | $\{a\}$ | $\{b\}$ |
|---|---|---|
| $\Gamma_P(M)$ | $\{a\}$ | $\{b\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\emptyset$ | $\emptyset$ |
| $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes |
| Is RSM ? | Yes | Yes |

*Example B.1.3.* Odd-Loop Over 3 Literals

$a \leftarrow\sim b$

$b \leftarrow\sim c$

$c \leftarrow\sim a$

| Candidate Min.Mod. M | $\{a, b\}$ | $\{b, c\}$ | $\{a, c\}$ |
|---|---|---|---|
| $\Gamma_P(M)$ | $\{b\}$ | $\{c\}$ | $\{a\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{a\}$ | $\{b\}$ | $\{c\}$ |
| $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 4$ | Yes, $\alpha = 4$ | Yes, $\alpha = 4$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes | Yes |
| Is RSM ? | Yes | Yes | Yes |

*Example B.1.4.* Even-Loop vs Odd-Loop

$c \leftarrow a, \sim c$

$a \leftarrow\sim b$

$b \leftarrow\sim a$

| Candidate Min.Mod. M | $\{b\}$ | $\{a, c\}$ |
|---|---|---|
| $\Gamma_P(M)$ | $\{b\}$ | $\{a\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\emptyset$ | $\{c\}$ |
| $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes |
| Is RSM ? | Yes | Yes |

*Example B.1.5.* Complex Odd-Loop

$a \leftarrow\sim a, \sim b$

$d \leftarrow\sim a$

$b \leftarrow d, \sim b$

| Candidate Min.Mod. M | $\{a\}$ | $\{b, d\}$ |
|---|---|---|
| $\Gamma_P(M)$ | $\emptyset$ | $\{d\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{a\}$ | $\{b\}$ |
| $\exists_{\alpha \geq 2} \Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes |
| Is RSM ? | Yes | Yes |

*Example B.1.6.* Illegal atom in the RAA set

$a \leftarrow\sim b$

$b \leftarrow\sim a$

$t \leftarrow a, b$

$k \leftarrow\sim t$

$i \leftarrow\sim k$

| Candidate Min.Mod. M | $\{a,k\}$ | $\{b,k\}$ | $\{a,t,i\}$ | $\{b,t,i\}$ |
|---|---|---|---|---|
| $\Gamma_P(M)$ | $\{a,k\}$ | $\{b,k\}$ | $\{a,i\}$ | $\{b,i\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\emptyset$ | $\emptyset$ | $\{t\}$ | $\{t\}$ |
| $\exists_{\alpha\geq 2}\Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ | No | No |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes | Yes | Yes |
| Is RSM ? | Yes | Yes | No | No |

*Example B.1.7.* Illegal non-minimal RAA set

$a \leftarrow\sim a$

$b \leftarrow\sim a$

$c \leftarrow\sim b$

$d \leftarrow\sim c$

| Candidate Min.Mod. M | $\{a,c\}$ | $\{a,b,d\}$ |
|---|---|---|
| $\Gamma_P(M)$ | $\{c\}$ | $\{d\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{a\}$ | $\{a,b\}$ |
| $\exists_{\alpha\geq 2}\Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | No |
| Is RSM ? | Yes | No |

*Example B.1.8.* Legal non-minimal RAA set

$a \leftarrow\sim b$

$b \leftarrow\sim a$

$c \leftarrow a, \sim c$

$c \leftarrow b, \sim c$

$d \leftarrow b, \sim d$

| Candidate Min.Mod. M | $\{a,c\}$ | $\{b,c,d\}$ |
|---|---|---|
| $\Gamma_P(M)$ | $\{a\}$ | $\{b\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{c\}$ | $\{c,d\}$ |
| $\exists_{\alpha\geq 2}\Gamma_P^\alpha(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes |
| Is RSM ? | Yes | Yes |

*Example B.1.9.* Intermixed OLONs

$a \leftarrow\sim b$

$b \leftarrow\sim c, e$

$c \leftarrow\sim a$

$e \leftarrow\sim e, a$

| Candidate Min.Mod. M | $\{a, c, e\}$ | $\{b, c\}$ | $\{a, b, e\}$ |
|---|---|---|---|
| $\Gamma_P(M)$ | $\{a\}$ | $\{c\}$ | $\{\}$ |
| $RAA_P(M) = M - \Gamma_P(M)$ | $\{c, e\}$ | $\{b\}$ | $\{a, b, e\}$ |
| $\exists_{\alpha \geq 2} \Gamma_P^{\alpha}(M) \supseteq RAA_P(M)$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ | Yes, $\alpha = 2$ |
| $RAA_P(M)$ is *Sustainable*? | Yes | Yes | Yes |
| Is RSM ? | Yes | Yes | Yes |

# Appendix C

# Source Code of the Implementations

Both implementations here shown were designed and built to cope with just simple rules of a Normal Logic Program, in the sense that there must not be any variables in the program. These implementations are thus prepared only for grounded NLPs.

More general and efficient implementations are already in the plans of our future work.

## C.1 The Meta-Interpreter for RSM

The implementation of the RSM Meta-Interpreter here presented runs under SWI-Prolog and, with minor modifications, it can be run under XSB-Prolog. This implementations consists on 2 files: "utils.P" which contains several useful predicates; and "`rsm_interpreter.P`" which contains the main predicates for top-down query answering based on Revised Stable Models.

We now show the contents of each of these files.

`rsm_interpreter.P:`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Revised Stable Models Top-Down Query Solver
%
% Coded by: Alexandre Miguel Pinto  amp@di.fct.unl.pt
% 2005/01
%
% Usage:
%
%   * predicate loadRSMProgram/1 loads the rules in a file. The rules
```

```
%     should be Normal Logic Program rules, i.e., without "not"s in the
%     heads of rules.
%
%     Rules take thus the form:    head <- lit1, not lit2, ..., litn
%
%   * predicate query/4 solves the query-list in the first argument
%     considering as true the literals in the second argument. Typically
%     the 2nd argument is an empty list in order not to influence the
%     derivation process. However, this 2nd argument can be used as a
%     mechanism for reasoning under hypothetical scenarios (possibly
%     counterfactual reasoning?). The third argument should be a
%     variable and it will contain the literals that were considered true
%     in the derivation process (a kind of mixture of abduction with
%     tabling). The fourth argument should also be a variable an it is
%     just for internal use - it keeps the list of ancestors during the
%     derivation process.
%
% This implementation is just a first prototype built to illustrate how
% the Revised Stable Models semantics allows top-down query answering.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%:- import append/3,member/2 from basics.
%:- dynamic hasRules/1, fact/1, rule/2.

:- op(900, fy, not).
:- op(1110, xfy, <-).
:- op(1200, xfx, :-).

:- ['utils.P'].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Loading of KB
%

init :- clean,
```

```
    touch(fact/1),
    touch(hasRules/1),
    touch(rule/2).

clean :-
    retractall(hasRules(_)),
    retractall(fact(_)),
    retractall(rule(_,_)).

loadRSMProgram(Filename) :-
    init,
    exists_file(Filename),
    see(Filename),
    loadRSMRules, !,
    seen.

loadRSMRules :- (repeat, continue, !) ; true.

continue :-
    (read(Rule),
    Rule \= end_of_file, !,
    loadRSMRule(Rule), !,
    fail);
    true.

loadRSMRule((Head <- Body)) :- !,
    my_assert(hasRules(Head)), !,
    body2list(Body, BodyList),
    my_assert(rule(Head, BodyList)), !.

loadRSMRule(Fact) :- !,
    my_assert(hasRules(Fact)), !,
    my_assert(fact(Fact)), !.

body2list((First, Rest), [First|LRest]) :- body2list(Rest, LRest).
body2list(Elem, [Elem]).
```

```
%
% Loading of KB
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Specific utilities
%

negLit(not Lit, not Lit) :- !.
negLit(Lit, not Lit).
negLit(not _).

posLit(not Lit, Lit) :- !.
posLit(Lit, Lit).
posLit(not _) :- !, fail. posLit(_).

toggle(Lit, TLit) :- posLit(Lit), !, negLit(Lit, TLit), !.
toggle(Lit, TLit) :- negLit(Lit), !, posLit(Lit, TLit), !.

%
% Specific utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% query
%

query([], A, A, _) :- !.
query([Lit|Rest], AbdsIn, AbdsOut, Ancestors) :-
    queryLit(Lit, AbdsIn, AbdsTemp, Ancestors), !,
    query(Rest, AbdsTemp, AbdsOut, Ancestors).

%
```

```
% query
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% queryLit
%

queryLit(Lit, AbdsIn, AbdsOut, _) :-
    ((fact(Lit), append(AbdsIn, [Lit], AbdsOut)) ;
    (member(Lit, AbdsIn), AbdsOut = AbdsIn)), !.
queryLit(Lit, Abds, _, _) :-
    toggle(Lit, TLit),
    member(TLit, Abds), !,
    fail.
queryLit(Lit, _, _, _) :-
    posLit(Lit),
    \+hasRules(Lit), !,
    fail.
queryLit(not Lit, AbdsIn, AbdsOut, _) :-
    \+ hasRules(Lit), !,
    append(AbdsIn, [not Lit], AbdsOut).
queryLit(Lit, AbdsIn, AbdsOut, Ancestors) :-
    posLit(Lit), !,
    append(Ancestors, [Lit], PossibleLoop), !,
    (loop(PossibleLoop, _, LoopType) ->
        (LoopType = positiveLoop ->
            (!, fail)
        ;
            append(AbdsIn, [Lit], AbdsOut), !
        ), !
    ;
        (rule(Lit, Body),
        append(Ancestors, [Lit], DeepAncestors),
        query(Body, AbdsIn, AbdsOut, DeepAncestors), !)).
```

```
queryLit(not Lit, AbdsIn, AbdsOut, Ancestors) :- !,
    append(Ancestors, [not Lit], PossibleLoop), !,
    (loop(PossibleLoop, _, LoopType) ->
        append(AbdsIn, [Lit], AbdsOut), !,
        (LoopType = evenLoop -> !, fail ; true), !
    ;
        rule(Lit, Body),
        append(Ancestors, [not Lit], DeepAncestors), !,
        \+ query(Body, AbdsIn, AbdsOut, DeepAncestors), !,
        append(AbdsIn, [not Lit], AbdsOut), !).

%
% queryLit
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Loop Detection
%

loop(Path, LitInLoop, LoopType) :- !,
    append(Begining, [LastLit], Path), !,
    posLit(LastLit, PosLastLit), !,
    negLit(LastLit, NegLastLit), !,
    (nth(PosLastLit, TempPIndex, Begining) ->
        (nth(NegLastLit, TempNIndex, Begining) ->
            min([TempPIndex, TempNIndex], Index)
        ;
            Index = TempPIndex
        )
    ;
        nth(NegLastLit, Index, Begining)
    ), !,
    LitInLoop = PosLastLit, !,
    get_sublist(Path, Index, last, Loop), !,
    get_loop_type(Loop, LoopType), !.
```

```
nnl([], 0).
nnl([not _|Rest], Number) :- !,
    nnl(Rest, NumberAux),
    Number is NumberAux + 1.
nnl([_|Rest], Number) :- !, nnl(Rest, Number).

get_loop_type(Loop, LoopType) :-
    nnl(Loop, NNLAux),
    Loop = [First|_],
    (negLit(First) ->
        NNL is NNLAux - 1
    ;
        NNL = NNLAux),
    (NNL = 0 ->
        LoopType = positiveLoop
    ;
        (even(NNL) ->
            LoopType = evenLoop
        ;
            LoopType = oddLoop)).

%
% Loop Detection
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

utils.P:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Aritmetic utilities
%

min([N], N).
min([F|R], F) :- min(R, X), F < X, !.
min([_|R], X) :- min(R, X).
```

```
max([N], N).
max([F|R], F) :- max(R, X), F > X, !.
max([_|R], X) :- max(R, X).

even(N) :- (X is N mod 2), X = 0.
odd(N) :- (X is N mod 2), X = 1.

%
% Aritmetic utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Assert / Retract utilities
%

my_assert(G) :- (\+ retract(G) -> assert(G) ; assert(G)).
my_retract(G) :- G, !, retract(G), my_retract(G).
my_retract(_).

touch(Predicate/Arity) :-
    functor(Term, Predicate, Arity),
    assert(Term),
    retract(Term).

%
% Assert / Retract utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% List utilities
%
```

```
is_sublist([],_).
is_sublist(_,[]) :- !, fail.
is_sublist([H|T], L) :- member(H,L), is_sublist(T,L).

get_sublist([], _, _, []).
get_sublist(L, S, last, SL) :-
    length(L, Length),
    get_sublist(L, S, Length, SL).

get_sublist(_, StartIndex, EndIndex, []) :-
    StartIndex > EndIndex, !.
get_sublist(List, StartIndex, EndIndex, SubList) :-
    StartIndex < 1, !,
    get_sublist(List, 1, EndIndex, SubList).
get_sublist(List, StartIndex, _, []) :-
    length(List, Length),
    StartIndex > Length, !.
get_sublist(List, StartIndex, EndIndex, SubList) :-
    length(List, Length),
    EndIndex > Length, !,
    get_sublist(List, StartIndex, Length, SubList).

get_sublist(List, I, I, [Elem]) :-
    nth(Elem, I, List), !.
get_sublist([First|Rest], 1, EndIndex, [First|SubRest]) :-
    NewEndIndex is EndIndex - 1,
    get_sublist(Rest, 1, NewEndIndex, SubRest).

get_sublist([_|Rest], StartIndex, EndIndex, SubList) :-
    NewStartIndex is StartIndex - 1,
    get_sublist(Rest, NewStartIndex, EndIndex, SubList).

nth(_, _, []) :- !, fail.
nth(E, 1, [E|_]) :- !.
nth(E, N, [X|R]) :-
    E \= X,
    nth(E, M, R),
```

```
    N is M + 1.

intersection([], _, []).
intersection(_, [], []).
intersection([Elem|Rest], List, [Elem|IRest]) :-
    member(Elem, List), !,
    intersection(Rest, List, IRest).
intersection([_|Rest], L2, L3) :- intersection(Rest, L2, L3).

union([], L, L).
union(L, [], L).
union([First|Rest], L2, URest) :-
    member(First, L2), !,
    union(Rest, L2, URest).
union([First|Rest], L2, [First|URest]) :- union(Rest, L2, URest).

subtract(L, [], L).
subtract([], _, []).
subtract([Elem|Rest], L2, L3) :-
    member(Elem, L2), !,
    subtract(Rest, L2, L3).
subtract([Elem|Rest], L2, [Elem|L3]) :- subtract(Rest, L2, L3).

num(X,[X|T],T).
num(X,[H|T],[H|B]) :- num(X,T,B).

get_subset([],_).
get_subset([H|T],A) :- num(H,A,B), get_subset(T,B).

sort_lists([], []).
sort_lists([First|Rest], [SFirst|SRest]) :-
    sort(First, SFirst),
    sort_lists(Rest, SRest).

powerset(Set, PS) :-
    setof(S, get_subset(S, Set), GPS),
    sort_lists(GPS, SGPS),
```

```
    remove_duplicates(SGPS, PS).

remove_duplicates([],[]).
remove_duplicates([E|L],R) :-
    member(E,L), !,
    remove_duplicates(L,R).

remove_duplicates([E|L],[E|R]) :-
remove_duplicates(L,R).

permutation([], []).
permutation(L1, L2) :-
    length(L1, TL),
    length(L2, TL),
    is_sublist(L1, L2),
    is_sublist(L2, L1).

reverse([], []) :- !.
reverse([E|R], L) :-
    reverse(R, RestRev), !,
    append(RestRev, [E], L), !.

%
% List utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% String utilities
%

string_replace('', _, _, '').
string_replace(String, Search, _, String) :-
    \+ sub_string(String, _, _, _, Search).

string_replace(String, Search, Replace, Result) :-
```

```
    string_concat(Begining, Rest, String),
    string_concat(Head, Search, Begining),
    string_concat(Head, Replace, NewBegining),
    string_replace(Rest, Search, Replace, RestResult),
    string_concat(NewBegining, RestResult, Result).

%
% String utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## C.2 The RSM Calculator

The implementation of the RSM Calculator is, like the top-dow implementation, based on 2 files: the "utils.P" file already shown in the previous section, and the following "`rsm_calculator.P`" file.

`rsm_calculator.P:`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Revised Stable Models Generator
%
% Coded by: Alexandre Miguel Pinto  amp@di.fct.unl.pt
% 2005/02
%
% Usage:
%
%   * predicate loadRSMProgram/1 is the same as in the top-down
%     query solver.
%
%   * predicate genRSMs/1 produces as output the list of Revised
%     Stable Models of the loaded RSM Program.
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%:- import append/3,member/2 from basics.
```

```
%:- import reverse/2 from lists.
%:- dynamic hasRules/1, fact/1, rule/2.

:- op(900, fy, not).
:- op(1110, xfy, <-).
:- op(1200, xfx, :-).

:- ['utils.P'].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Loading of KB
%

init :-
    op(900, fy, not),
    op(1110, xfy, <-),
    op(1200, xfx, :-),
    ['utils.P'],
    clean,
    touch(fact/1),
    touch(hasRules/1),
    touch(rule/2),
    touch(lit/1).

clean :-
    retractall(hasRules(_)),
    retractall(fact(_)),
    retractall(rule(_,_)),
    retractall(lit(_)).

loadRSMProgram(Filename) :-
    init,
    exists_file(Filename),
    see(Filename),
    loadRSMRules, !,
    seen.
```

```
loadRSMRules :- (repeat, continue, !) ; true.

continue :-
    (read(Rule),
    Rule \= end_of_file, !,
    loadRSMRule(Rule), !,
    fail);
    true.

loadRSMRule((Head <- Body)) :- !,
    my_assert(hasRules(Head)), !,
    my_assert(lit(Head)), !,
    body2list(Body, BodyList), !,
    my_assert_lits(BodyList), !,
    my_assert(rule(Head, BodyList)), !.

loadRSMRule(Fact) :- !,
    my_assert(hasRules(Fact)), !,
    my_assert(fact(Fact)), !,
    my_assert(lit(Fact)), !.

body2list((First, Rest), [First|LRest]) :- body2list(Rest, LRest).
body2list(Elem, [Elem]).

my_assert_lits([]). my_assert_lits([not Lit|Rest]) :- !,
    my_assert(lit(Lit)), !,
    my_assert_lits(Rest).

my_assert_lits([Lit|Rest]) :- !,
    my_assert(lit(Lit)), !,
    my_assert_lits(Rest).

%
% Loading of KB
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% RSM Detector
%

is_sm(Model) :- raa(Model, []), !.

is_rsm(Model) :-
    raa(Model, RAA), !,
    (RAA = [] -> % It's a Stable Model!
        true, !  % So, it's a Revised Stable Model
    ;
        % If it's not a Stable Model
        sustainable(RAA),!,
        % Check if it complies with the 3rd condition
        gammaNsupseteqRAA(Model, RAA, _)
        % and with the 2nd condition of the RSM definition
    ).

    sustainable([]) :- !.
    sustainable([_]) :- !.
    sustainable(L) :- forall(member(X,L), sustainability_implication(X,L)).

    sustainability_implication(X,[]) :- !.
    sustainability_implication(X,L) :-
        delete(L,X,L_X),
        (\+ sustainable(L_X) ;
         add_facts(L_X), !,
         wfm(WFM_P_U_L_X), !,
         gamma(WFM_P_U_L_X, TrueOrUndefOfP_U_L_X), !,
         member(X, TrueOrUndefOfP_U_L_X)).

add_facts([]) :- !. add_facts([Fact|RestFacts]) :-
    my_assert(fact(Fact)), !,
    add_facts(RestFacts).
```

```
remove_added_facts([]) :- !.
remove_added_facts([Fact|RestFacts]) :-
    my_retract(fact(Fact)), !,
    remove_added_facts(RestFacts).

%
% RSM Detector
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Generate Revised Stable Models
%

genRSMs(RSMs) :-
    genMMs(MMs), !,
    % RSMs are Minimal Models - generate
    % the possible candidates: the MMs
    findRSMs(MMs, RSMs), !.

findRSMs(MMs, RSMs) :-
    reverse(MMs, RMMs), !,
    buildRSMs(RMMs, RSMsAux), !,
    reverse(RSMsAux, RSMs), !.

buildRSMs([], []) :- !.
buildRSMs([MM|RestMMs], RSMs) :-
    (is_rsm(MM, RestMMs) ->
        buildRSMs(RestMMs, RestRSMs), !,
        append([MM], RestRSMs, RSMs), !
    ;
        buildRSMs(RestMMs, RSMs)
    ).

%
% Generate Revised Stable Models
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Generate Candidate Minimal Models
%

genMMs(MMs) :-
    setof(Lit, lit(Lit), Lits), !,
    powerset(Lits, Interpretations), !,
    getCMs(Interpretations, CMs), !,
    minimize(CMs, MMs), !.

getCMs([], []) :- !.
getCMs([Interpretation|RestInterpretations], CMs) :-
    (is_CM(Interpretation) -> % Interpretation is a Classical Model
        getCMs(RestInterpretations, RestCMs), !,
        append([Interpretation], RestCMs, CMs), !
    ;
        getCMs(RestInterpretations, CMs), !).

is_CM(Interpretation) :-
    program_division(Interpretation, Rules), !,
    iterate_Tp(Interpretation, Rules, Result), !,
    (setof(Fact, fact(Fact), Fs) ->
        Facts = Fs
    ;
        Facts = []
    ), !,
    union(Facts, Result, TPResult), !,
    is_sublist(TPResult, Interpretation), !.

minimize([], []) :- !.
minimize(CMs, MMs) :-
    append(Before, [LastCM], CMs), !,
    (is_minimal(LastCM, Before) ->
```

```
        minimize(Before, OtherMMs), !,
        append(OtherMMs, [LastCM], MMs), !
    ;
        minimize(Before, MMs)), !.

is_minimal(_, []) :- !.
is_minimal(Set, [FirstSet|RestSetList]) :- !,
    \+ is_sublist(FirstSet, Set), !,
    is_minimal(Set, RestSetList), !.

%
% Generate Candidate Minimal Models
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% WFM
%

wfm(WFM) :- iterateGamma2ToFixpoint([], WFM).

iterateGamma2ToFixpoint(LastGamma2, Fixpoint) :-
    gamma2(LastGamma2, NewGamma2), !,   % apply Gamma2
    (LastGamma2 = NewGamma2 -> % If it is a fixpoint
        Fixpoint = LastGamma2   % then stop
    ;
        iterateGamma2ToFixpoint(NewGamma2, Fixpoint)).
        % else continue Gamma2 iterations

%
% WFM
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% Gamma^N
%

gamma(Model, 0, Model) :- !.
gamma(Model, 1, Gamma) :- !,
    gamma(Model, Gamma), !.
gamma(Model, N, GammaN) :- !,
    gamma(Model, Gamma), !,
    M is N - 1, !,
    gamma(Gamma, M, GammaN), !.

%%%%%%%%
% Gamma2
gamma2(Model, Gamma2) :- !, gamma(Model, 2, Gamma2), !.
% Gamma2
%%%%%%%%

%
% Gamma^N
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Gamma^N >= RAA_P(M) ?
%

gammaNsupseteqRAA(Model, RAA, N) :-
    gamma(Model, FirstGamma), !,
    gamma(FirstGamma, SecondGamma), !,
    anyGammaSupseteqRAA([SecondGamma], RAA, M), !,
    N is M + 1.

anyGammaSupseteqRAA(GammasList, RAA, M) :-
    append(_, [LastGamma], GammasList), !,
    (is_sublist(RAA, LastGamma) ->
        length(GammasList, M)
```

```
    ;
        gamma(LastGamma, NewGamma), !,
        (member(NewGamma, GammasList) ->
            !, fail, !
        ;
            append(GammasList, [NewGamma], NewGammasList), !,
            anyGammaSupseteqRAA(NewGammasList, RAA, M), !
        )
    ).

%
% Gamma^N >= RAA_P(M) ?
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Gamma^N Fixpoint
%

gammaNfixpoint(Interpretation, N, Fixpoint) :-
    gammaNfixpoint(Interpretation, [], N, Fixpoint), !.
gammaNfixpoint(Interpretation, [], N, Fixpoint) :- !,
    gammaNfixpoint(Interpretation, [Interpretation], N, Fixpoint), !.
gammaNfixpoint(Interpretation, GammasList, N, Fixpoint) :-
    append(RestBefore, [LastGamma], GammasList), !,
    (nth(LastGamma, Index, RestBefore) ->
        N is Index - 1,
        Fixpoint = LastGamma
    ;
        gamma(LastGamma, NewGamma), !,
        append(GammasList, [NewGamma], NewGammasList), !,
        gammaNfixpoint(Interpretation, NewGammasList, N, Fixpoint), !), !.

%
% Gamma^N Fixpoint
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% RAA Set
%

raa(Model, RAA) :-
    gamma(Model, Gamma), !,
    is_sublist(Gamma, Model), !,
    % Any Classical Model is greater than or equal to its Gamma
    subtract(Model, Gamma, RAA), !.

%
% RAA Set
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Gamma Operator
%
gamma(Interpretation, Result) :-
    program_division(Interpretation, Rules), !,
    iterate_Tp_omega(Rules, Result).

program_division(Interpretation, Rules) :-
    get_alive_rules(Interpretation, AliveRules), !,
    remove_defaults(AliveRules, Rules), !.

get_alive_rules([], AliveRules) :- !,
    setof(rule(H,B), rule(H,B), AliveRules), !.

get_alive_rules([Lit|Rest], AliveRules) :-
    setof(rule(H,B), rule(H,B), AllRules), !,
    get_remaining_alive_rules([Lit|Rest], AllRules, AliveRules), !.
```

```
get_remaining_alive_rules([], AllRules, AllRules) :- !.
get_remaining_alive_rules([Lit|Rest], AllRules, AliveRules) :-
    remove_rules_with_not_Lit(AllRules, Lit, RemainingRules), !,
    get_remaining_alive_rules(Rest, RemainingRules, AliveRules).

remove_rules_with_not_Lit([], _, []) :- !.
remove_rules_with_not_Lit([Rule|RestRules], Lit, RemainingRules) :-
    Rule = rule(_, B),
    member(not Lit, B), !,
    remove_rules_with_not_Lit(RestRules, Lit, RemainingRules), !.

remove_rules_with_not_Lit([Rule|RestRules], Lit, RemainingRules) :- !,
    remove_rules_with_not_Lit(RestRules, Lit, RemainingRulesAux), !,
    append([Rule], RemainingRulesAux, RemainingRules), !.

remove_defaults([], []) :- !.
remove_defaults([Rule|RestRules], Rules) :-
    Rule = rule(H, B),
    remove_defaults_from_body(B, PB), !,
    remove_defaults(RestRules, OtherRules), !,
    append([rule(H, PB)], OtherRules, Rules).

remove_defaults_from_body([], []) :- !.
remove_defaults_from_body([not _|Rest], PB) :- !,
    remove_defaults_from_body(Rest, PB), !.

remove_defaults_from_body([Lit|Rest], PB) :- !,
    remove_defaults_from_body(Rest, PBAux), !,
    append([Lit], PBAux, PB), !.

iterate_Tp_omega([], Result) :- !,
    (setof(Fact, fact(Fact), Facts) ->
        Result = Facts
    ;
        Result = []), !.

iterate_Tp_omega(Rules, Result) :- !,
```

```
    (setof(Fact, fact(Fact), Fs) ->
        Facts = Fs
    ;
        Facts = []), !,
    iterate_Tp(Facts, Rules, Result), !.

iterate_Tp(Facts, [], Facts) :- !.
iterate_Tp(Facts, Rules, Result) :-
    tp(Facts, Rules, TempResult), !,
    (Facts = TempResult ->
        Result = Facts
    ;
        iterate_Tp(TempResult, Rules, Result)
    ) ,!.

tp(Facts, Rules, Result) :-
    remove_facts_from_bodies(Facts, Rules, CleanRules), !,
    get_heads_of_empty_rules(CleanRules, Heads), !,
    union(Facts, Heads, Result), !.

remove_facts_from_bodies([], Rules, Rules) :- !.
remove_facts_from_bodies(_, [], []) :- !.
remove_facts_from_bodies([Fact|RestFacts], Rules, CleanRules) :-
    remove_fact_from_bodies(Fact, Rules, AuxCleanRules), !,
    remove_facts_from_bodies(RestFacts, AuxCleanRules, CleanRules), !.

remove_fact_from_bodies(_, [], []) :- !.
remove_fact_from_bodies(Fact, [Rule|RestRules], [CleanRule|RestCleanRules]) :-
    remove_fact_from_body(Fact, Rule, CleanRule), !,
    remove_fact_from_bodies(Fact, RestRules, RestCleanRules), !.

remove_fact_from_body(Fact,Rule,rule(H, CleanBody)) :-
    Rule = rule(H, B), !,
    subtract(B, [Fact], CleanBody), !.

get_heads_of_empty_rules([], []) :- !.
get_heads_of_empty_rules([rule(H, [])|RestCleanRules], Heads) :- !,
```

```
    get_heads_of_empty_rules(RestCleanRules, RestHeads), !,
    append([H], RestHeads, Heads).

get_heads_of_empty_rules([rule(_, _)|RestCleanRules], Heads) :- !,
    get_heads_of_empty_rules(RestCleanRules, Heads), !.

%
% Gamma Operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Specific utilities
%

negLit(not Lit, not Lit) :- !.
negLit(Lit, not Lit).
negLit(not _).

posLit(not Lit, Lit) :- !.
posLit(Lit, Lit).
posLit(not _) :- !, fail.
posLit(_).

toggle(Lit, TLit) :- posLit(Lit), !, negLit(Lit, TLit), !.
toggle(Lit, TLit) :- negLit(Lit), !, posLit(Lit, TLit), !.

%
% Specific utilities
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

# Appendix D

# Tests and results of the RSM Implementations

For testing the implementations of the top-down proof-procedure for query solving, and of the RSM calculator the following set of NLPs was used.

```
P1 =

    a <- not b.
    b <- not a.
    c <- a, not c.
    c <- b, not c.
    d <- b, not d.


P2 =

    a <- not b.
    b <- not a.
    t <- a, b.
    k <- not t.
    i <- not k.


P3 =

    a <- not a.
```

```
    b <- not a.
    c <- not b.
    d <- not c.
```

```
P4 =
```

```
    a.
    b <- not c.
    d <- e.
```

Using the test program $P_1$ above we obtained the following results

| Query | Result |
|---|---|
| query([a],[],_,_) | Yes |
| query([b],[],_,_) | Yes |
| query([c],[],_,_) | Yes |
| query([d],[],_,_) | Yes |
| query([a,b],[],_,_) | No |
| query([a,c],[],_,_) | Yes |
| query([a,d],[],_,_) | No |
| query([b,c],[],_,_) | Yes |
| query([b,d],[],_,_) | Yes |
| query([b,c,d],[],_,_) | Yes |

```
?- genRSMs(RSMs)
RSMs = [[a, c], [b, c, d]]
```

For the test program $P_2$ the results were

| Query | Result |
|---|---|
| query([a],[],_,_) | Yes |
| query([b],[],_,_) | Yes |
| query([t],[],_,_) | No |
| query([k],[],_,_) | Yes |
| query([i],[],_,_) | No |
| query([a,k],[],_,_) | Yes |
| query([a,t],[],_,_) | No |
| query([a,b],[],_,_) | No |
| query([a,i],[],_,_) | No |
| query([k,i],[],_,_) | No |
| query([k,t],[],_,_) | No |

```
?- genRSMs(RSMs)
RSMs = [[a, k], [b, k]]
```

With the test program $P_3$ the results were

| Query | Result |
|---|---|
| query([a],[],_,_) | Yes |
| query([b],[],_,_) | No |
| query([c],[],_,_) | Yes |
| query([d],[],_,_) | No |
| query([a,b,d],[],_,_) | No |
| query([a,c],[],_,_) | Yes |

```
?- genRSMs(RSMs)
RSMs = [[a, c]]
```

Finally, the test program $P_4$ gave the following results

| Query | Result |
|---|---|
| query([a],[],_,_) | Yes |
| query([b],[],_,_) | Yes |
| query([c],[],_,_) | No |
| query([d],[],_,_) | No |

```
?- genRSMs(RSMs)
RSMs = [[a, b]]
```

All the results here presented correspond precisely to the ones expected under the Revised Stable Models semantics, both under the top-down proof-procedure query-solver, and the RSM calculator.