José Júlio Alves Alferes

Semantics of Logic Programs with Explicit Negation

Dissertação apresentada para a obtenção do Grau de Doutor em Engenharia Informática, especialidade Inteligência Artificial, pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(1993)

Acknowledgements

First of all, I would like to thank my parents, Acácio and Dionísia, for the incentive they gave me to pursue the goal of making the PhD, their encouragement, support and many other reasons.

Also special thanks to Cristina for her support, and especially for bearing with me during the period while this thesis was being written.

Special thanks to my supervisor, Luís Moniz Pereira, for lots of reasons including but not limited to: his availability for discussions, even when he had lots of other urgent things to do; the valuable comments and advice he gave me about this work; for introducing me to the right researchers at the right time, giving me the chance of learning a lot with those researchers; for providing for the good working conditions I had during the preparation of the PhD thesis; for teaching me how to do scientific research in its many aspects, and for the pleasure of doing it together with him.

I would also like to thank my colleagues Joaquim Nunes Aparício and Carlos Damásio, who collaborated with me in many tasks during the three years of preparation of this thesis.

For many profitable discussions, thanks are due to, among others, my colleagues Salvador Abreu, Luís Caires, Gabriel David, Luís Monteiro, António Porto, and Irene Rodrigues.

On the international side, I would like to thank, for valuable comments and discussions, Marc Denecker, Michael Gelfond, Tony Kakas, Vladimir Lifschitz, Paolo Mancarella, David Peirce, Halina Przymusinska, Teodor Przymusinski, and Gerd Wagner. Special thanks to Phan Minh Dung.

I'm indebted to Carlos Damásio and Luís Monteiro, who kindly accepted to read a preliminary version of this report and give me their comments, and to Salvador Abreu for helping me with the French "sommaire".

I would like to acknowledge JNICT (Junta Nacional de Investigação Científica e Tecnológica) for giving me the PhD grant (no. 137/90-IA) that paid for my survival during these last three years.

I thank Esprit BR projects Compulog (no. 3012) and Compulog 2 (no. 6810) for their support, and for giving me the chance of knowing many european researchers in the area of logic programming, with whom I had many aclarifying discussion.

For giving me working conditions, I acknowledge the Departamento de Informática of Faculdade de Ciências e Tecnologia of Universidade Nova de Lisboa, the Centro de Inteligência Artificial (CRIA) of Uninova, and Centro de Informática da Universidade Nova de Lisboa (CIUNL).

I'm indebted to Anabela Rodrigues and Filipa Reis, for managing most of my bureaucracy, and arranging my travels, during the period of preparation of this thesis and even before.

Abstract

After a historical introduction, the bulk of the thesis concerns the study of a declarative semantics for logic programs.

The main original contributions are:

- *WFSX* (Well–Founded Semantics with eXplicit negation), a new semantics for logic programs with explicit negation (i.e. extended logic programs), which compares favourably in its properties with other extant semantics.
- A generic characterization schema that facilitates comparisons among a diversity of semantics of extended logic programs, including *WFSX*.
- An autoepistemic and a default logic corresponding to *WFSX*, which solve existing problems of the classical approaches to autoepistemic and default logics, and clarify the meaning of explicit negation in logic programs.
- A framework for defining a spectrum of semantics of extended logic programs based on the abduction of negative hypotheses. This framework allows for the characterization of different levels of scepticism/credulity, consensuality, and argumentation. One of the semantics of abduction coincides with *WFSX*.
- O–semantics, a semantics that uniquely adds more CWA hypotheses to *WFSX*. The techniques used for doing so are applicable as well to the well–founded semantics of normal logic programs.
- By introducing explicit negation into logic programs contradiction may appear. I present two approaches for dealing with contradiction, and show their equivalence. One of the approaches consists in avoiding contradiction, and is based on restrictions in the adoption of abductive hypotheses. The other approach consists in removing contradiction, and is based in a transformation of contradictory programs into noncontradictory ones, guided by the reasons for contradiction.

Sumário

Depois de uma breve introdução histórica, o grosso da tese consiste no estudo de semânticas declarativas de programas em lógica.

As principais contribuições originais são:

- A *WFSX* (semântica bem–fundada com negação explícita), uma nova semântica para programas em lógica com negação explícita (i.e. programas em lógica extendidos), que é melhor que outras semântica existentes, nas suas propriedades.
- Um esquema genérico de caracterização que facilita comparações entre uma diversidade de semânticas, incluindo a *WFSX*.
- Uma lógica auto–epistémica e uma lógica de regras por omissão correspondentes à *WFSX*, que por um lado resolvem problemas das abordagens clássicas a lógicas auto–epistémicas e a lógicas de regras por omissão, e por outro clarificam o significado da negação explícita em programas em lógica.
- Um enquadramento de semânticas para programas em lógica extendidos com base na abdução de hipóteses negativas. Este enquadramento permite a caracterização de diferentes graus de cepticismo/credulidade, consensualidade e argumentação. Uma das semânticas de abdução coincide com a *WFSX*.
- A "semântica O", uma semântica que acrescenta à *WFSX* hipóteses não contraditáveis. As técnicas usadas para a definição desta semântica são também aplicáveis à semântica bem–fundada de programas normais.
- Com a introdução da negação explícita põe–se a questão do tratamento da contradição. Introduzem–se duas abordagens, que se mostram equivalentes, para lidar com a contradição. Uma consiste em evitá–la e a outra em removê–la, e são tratadas respectivamente através de restrições na adopção de hipóteses abdutivas, e da transformação de programas contraditórios em programas não contraditórios, guiada pelas razões da contradição.

Sommaire

Après une brève introduction historique, le gros de la thèse consiste en une étude de sémantiques déclaratives de la programmation logique.

Les principales contribution originales sont:

- La *WFSX* (sémantique bien fondée avec de la négation explicite), une nouvelle sémantique pour les programmes logiques avec la négation explicite (c.a.d. programmes logiques étendus), laquelle est meilleure que d'autres sémantiques existantes, en virtu de ses proprietés.

- Un schéma caractérisant générique qui permet la comparison entre une variété de sémantiques, y compris la *WFSX*.

- Une logique auto–épistémique et une logique de régles à défaut correspondantes à la *WFSX*, qui d'un coté résoudent quelques problèmes des abordages classiques aux logiques épistémiques et des logiques de régles à défaut, et qui d'autre part clarifient la signification de la négation explicite en programmation logique.

- Un cadre pour de sémantiques de la programmation logique étendue, fondé sur l'abduction d'hypothèses négatives. Ce cadre va permettre la caractérisation de plusieurs degrés des cepticisme/crédulité, de consensualité et d'argumentation. Une de ces sémantiques de l'abduction coincide avec la *WFSX*.

- L'O–sémantique, une sémantique qui ajoute à la *WFSX* des hypothéses non–contraditables. Les techniques emploiyées pour la définition de cette sémantique sont aussitôt valables pour la sémantique bien fondée de programmes normaux.

- Avec l'introduction de la négation explicite il se pose la question du traitement de la contradiction. On introduit deux approches, qui se révelent tout à fait équivalens, pour bien faire face à la contradiction. L'une l'entre elles consiste à l'éviter et l'autre à la dèfaire. Ils sont ateints, respéctivement, soit par des restrictions sur l'adoption des hypothéses abductives, soit par une transformation de programmes contradictoires en d'autres programmes non contradictoires, qui est guidée par les causes mêmes de la contradiction.

Contents

List of Figures

Preface

This work is divided into two quite distinct parts: the first makes a brief historical overview of the field of logic programming semantics; the second presents the original contributions of the thesis in this field. For the sake of completeness I present, in appendix A, a Prolog top–down interpreter for the semantics *WFSX*; additional definitions needed for some proofs are presented in appendix B; finally, appendix C, cointains the proofs of theorems that, for the sake of continuity, were not inserted along in the text.

The aim of the first part is to sensitize the reader to the issue of logic programming semantics, provide background and notation, and make clear the state of the art in the area at the inception of the work reported in this thesis.

In chapter 1, I begin by defining the language of normal logic programs. Then I briefly describe several approaches to the semantics of normal programs, and their treatment of negation as failure. Special attention is given to the stable models and well–founded semantics, for which I present the formal definitions.

In chapter 2, I start by providing some motivation for extended logic programs, i.e. normal logic programs extended with explicit negation, and define their language. Next, I present several extant semantics for such programs.

The structure of the second part, containing the original contributions, is as follows:

I begin, in chapter 3, with the motivation for a new semantics of extended logic programs. There, I point out why I'm not completely satisfied with other present–day semantics, and proffer some intuitively appealing properties a semantics should comply with.

In chapter 4, I expound *WFSX*, a semantics for extended logic programs that subsumes the well founded semantics of normal programs. I begin by providing definitions of interpretation and model, for programs extended with explicit negation. Next I introduce the notion of stability in models, and use it to define the *WFSX*. Finally, some of its properties are examined, with special incidence on those concerning its existence.

The first part of chapter 5 is devoted to contrasting and characterizing a variety of semantics for extended logic programs, including *WFSX*, in what concerns their use of and their meaning ascribed to a second kind of negation, and how the latter is related to both classical negation and the default negation (or negation as failure). For this purpose I define a parametrizeable schema to characterize and encompass a diversity of proposed semantics for extended logic programs. In the second part of that chapter, and based on the similarities between the parametrizable schema and the definitions of autoepistemic logics, I proceed to examine the relationship between the latter and extended logic programs. By doing so, an epistemic meaning of the second kind of negation is extracted. The relationship results clarify the use of logic programs for representing knowledge and belief.

Chapter 6 presents a semantics for default theories, and shows its rapport with *WFSX*. First I point out some issues faced by semantics for default theories, and identify some basic principles a default theory semantics should enjoy. Second, I present a default semantics that resolves the issues whilst respecting the principles (which other semantics don't). Afterwards

I prove the close correspondence between default theories under such a semantics and *WFSX*. Based on this correspondence result, I supply an alternative definition of *WFSX* not relying on 3–valued logic but instead on 2–valued logic alone.

Subsequently, in chapter 7, I characterize a spectrum of more or less sceptical and credulous semantics for extended logic programs, and determine the position of *WFSX* in this respect. I do so by means of a coherent, flexible, unifying, and intuition appealing framework for the study of explicit negation in logic programs, based on the notion of admissible scenaria. The main idea of the framework is to consider default literals as abducibles, i.e. they can be hypothesized. In the same chapter I also bring out the intimate relationship between this approach and argumentation systems.

With the introduction of explicit negation into logic programs contradiction may arise. In chapter 8, I put forth two approaches for dealing with contradiction: one persists in avoiding it, based on a generalization of the framework of chapter 7, whereby additional restrictions on the adoption of abductive hypotheses are imposed; the other approach consists in removing contradiction, and relies on a transformation of contradictory programs into noncontradictory ones, guided by the reasons for contradiction. Moreover I show that the contradiction avoidance semantics of a program P is equivalent to the *WFSX* of the program resulting from P by transforming it according to the contradiction removal methods.

As deployed in chapter 4, *WFSX* is based on a generalization of the well–founded semantics for normal logic programs. However it can be argued, and this is carried out in chapter 9, that sometimes the well–founded semantics is overly careful in deciding about the falsity of some atoms, leaving them undefined, and that a suitable form of closed world assumption can be employed to safely and undisputably assume false some additional atoms, otherwise absent from the well–founded model of a program. This provides a new semantics for normal programs – the O–semantics. I then proceed to generalize the O–semantics to extended logic programming and, finally, I compare it to *WFSX*.

Lastly, in chapter 10, I produce additional properties of *WFSX*, including complexity, and make further comparisons with other semantics on the basis of those properties (which are essentially structural in nature).

The best way to read this thesis is by reading the chapters in the sequence they appear in. However, if the reader is not interested in the whole work, or is more keen on some issues rather than others, alternative reading paths are possible; they are shown in figure 0.1.

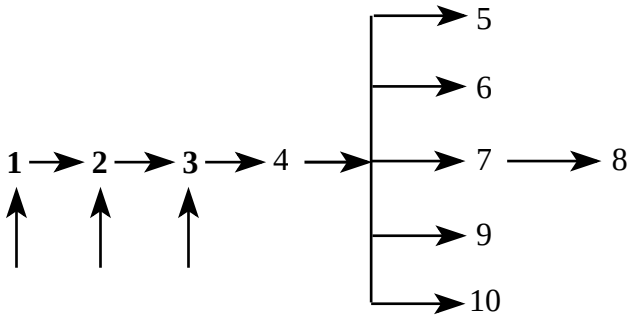

Figure 0.1: Reading paths, and three possible entry points.

If you are familiarized with the issue of extended logic programs semantics you can skip the first part, i.e. chapters 1 and 2.

If you are familiarized with the issue of normal logic programs semantics, but not with

explicit negation, you can skip chapter 1 and start with chapter 2.

Otherwise, you should start by reading the first part.

The table below indicates, for different possible interests, the corresponding reading paths of figure 0.1:

Definition of *WFSX*	3 – 4
Extended logic programs and autoepistemic logics	3 – 4 – 5
Extended logic programs and default logic	3 – 4 – 6
Extended logic programs abduction, and argumentation systems	3 – 4 – 7
Extended logic programs and belief revision	3 – 4 – 7 – 8
Semantics assuming more negative hypotheses	3 – 4 – 9
WFSX, its structural properties, and complexity	3 – 4 – 10

Quite a few of the results presented in this thesis rely much on joint European projects' work, mostly with my thesis supervisor Luís Moniz Pereira, but also with my colleague Joaquim Nunes Aparício, and Phan Minh Dung from the Asian Institute of Technology. At the beginning of each chapter I mention the joint work, with bearing on its subject matter, that we published in the three years of preparation of this thesis.

Part I

Semantics of Logic Programs: A Brief Historical Overview

Computational Logic arose from the work, begun by logicians in the 1950's, on the automation of logical deduction, and was fostered in the 1970's by Colmerauer *et al.* [Colmerauer *et al.*, 1973] and Kowalski [Kowalski, 1974, Kowalski, 1979] as Logic Programming. It introduced to computer science the important concept of *declarative* – as opposed to *procedural* – programming. Ideally, a programmer should only be concerned with the declarative meaning of his program, while the procedural aspects of program's execution are handled automatically. The Prolog language [Colmerauer *et al.*, 1973] became the privileged vehicle approximating this ideal. The first Prolog compiler [Warren *et al.*, 1977] showed that it could be a practical language and disseminated it worldwide.

The developments of formal foundations of logic programming began in the late 1970's especially with the works [Emden and Kowalski, 1976, Clark, 1978, Reiter, 1978]. Further progress in this direction was achieved in the early 1980's, leading to the appearance of the first book on the foundations of logic programming [Lloyd, 1984]. The selection of logic programming as the underlying paradigm for the Japanese Fifth Generation Computer Systems Project led to the rapid proliferation of various logic programming languages.

Due to logic programming's declarative nature, it quickly became a candidate for knowledge representation. Its adequateness became more apparent after the relationships established in the mid 1980's between logic programs and deductive databases [Reiter, 1984, Gallaire *et al.*, 1984, Lloyd and Topor, 1985, Lloyd and Topor, 1986, Minker, 1988].

The use of both logic programming and deductive databases for knowledge representation is based on the so called *"logical approach to knowledge representation"*. This approach rests on the idea of providing machines with a logical specification of the knowledge that they possess, thus making it independent of any particular implementation, context–free, and easy to manipulate and reason about.

Consequently, a precise meaning (or semantics) must be associated with any logic program in order to provide its declarative specification. The performance of any computational mechanism is then evaluated by comparing its behaviour to the specification provided by the declarative semantics. Finding a suitable declarative semantics for logic programs has been acknowledged as one of the most important and difficult research areas of logic programming.

In this part we make a quick historical overview of the results in the last 15 years in the area of logic program's declarative semantics. This overview is divided into two chapters. In the first we review some of the most important semantics of normal logic programs. In the second we motivate the need of extending logic programming with a second kind of negation, and overview recent semantics for such extended programs.

Chapter 1

Normal logic programs

Several recent overviews of normal logic programming semantics can be found in the literature (e.g. [Shepherdson, 1988, Shepherdson, 1990, Przymusinska and Przymusinski, 1990, Monteiro, 1992, Apt and Bol, 1993]). Here, for the sake of this text's self–sufficiency and to introduce some motivation, we distill a brief overview of the subject. In some parts we follow closely the overview of [Przymusinska and Przymusinski, 1990].

The structure of the chapter is as follows: first we present the language of normal logic programs and give some definitions needed in the sequel. Then we briefly review the first approaches to the semantics of normal programs and point out their problems. Finally, we expound in greater detail two more recent proposals, namely stable models and well–founded semantics.

1.1 Language

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_is 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_is are terms. A literal is either an atom A or its negation *not* A. We dub default literals those of the form *not* 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:

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

where H is an atom and each of the L_is 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.

In this work we restrict ourselves to Herbrand interpretations and models[1]. Thus, without loss of generality (cf. [Przymusinska and Przymusinski, 1990]), we coalesce a normal logic program P with its grounded version.

1.1.1 Interpretations and models

Next we define 2 and 3–valued Herbrand interpretations and models of normal logic programs. Since non–Herbrand interpretations are beyond the scope of this work, in the sequel we sometimes drop the qualification Herbrand.

Definition 1.1.1 (2–valued interpretation) *A 2–valued interpretation I of a normal logic program P is any subset of the Herbrand base $\mathcal{H}$ of P.*

Clearly, any 2–valued interpretation I can be equivalently viewed as a set

$$T \cup not\ F\ ^{2}$$

where $T = I$ and is the set of atoms which are true in I, and $F = \mathcal{H} - T$ is the set of atoms which are false in I. These interpretations are called 2–valued because in them each atom is either true or false, i.e. $\mathcal{H} = T \cup F$.

As argued in [Przymusinska and Przymusinski, 1990], interpretations of a given program P can be thought of as "possible worlds" representing possible states of our knowledge about the meaning of P. Since that knowledge is likely to be incomplete, we need the ability to describe interpretations in which some atoms are neither true nor false but rather undefined, i.e. we need 3–valued interpretations:

Definition 1.1.2 (3–valued interpretation) *By a 3–valued interpretation I of a program P we mean a set*

$$T \cup not\ F$$

where T and F are disjoint subsets of the Herbrand base $\mathcal{H}$ of P.

The set T (the T-part of I) contains all ground atoms true in I, the set F (the F-part of I) contains all ground atoms false in I, and the truth value of the remaining atoms is unknown (or undefined).

It is clear that 2–valued interpretations are a special case of 3–valued ones, for which $\mathcal{H} = T \cup F$ is additionally imposed.

Proposition 1.1.1 *Any interpretation $I = T \cup not\ F$ can equivalently be viewed as a function $I : \mathcal{H} \to V$ where $V = \left\{0, \frac{1}{2}, 1\right\}$, defined by:*

$$I(A) = \begin{cases} 0 & \text{if} \quad not\ A \in I \\ 1 & \text{if} \quad A \in I \\ \frac{1}{2} & \quad otherwise \end{cases}$$

Of course, for 2–valued interpretations there is no atom A such that $I(A) = \frac{1}{2}$.

Models are defined as usual, and based on a truth valuation function:

Definition 1.1.3 (Truth valuation) *If I is an interpretation, the truth valuation $\hat{I}$ corresponding to I is a function $\hat{I} : C \to V$ where C is the set of all formulae of the language, recursively defined as follows:*

[1] For the subject of semantics based on non–Herbrand models, and solutions to the problems resulting from always keeping Herbrand models see e.g. [Kunen, 1987, Przymusinski, 1989b, Gelder *et al.*, 1991].

[2] Where $not\ \{a_1, \ldots, a_n\}$ stands for $\{not\ a_1, \ldots, not\ a_n\}$.

- *if A is a ground atom then $\hat{I}(A) = I(A)$.*
- *if S is a formula then $\hat{I}(not\ S) = 1 - \hat{I}(S)$.*
- *if S and V are formulae then*
  - $\hat{I}((S,V)) = min(\hat{I}(S), \hat{I}(V))$.
  - $\hat{I}(V \leftarrow S) = 1$ *if* $\hat{I}(S) \leq \hat{I}(V)$, *and* 0 *otherwise.*

Definition 1.1.4 (3–valued model) *A 3–valued interpretation I is called a 3–valued model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.*

The special case of 2–valued models has the following straightforward definition:

Definition 1.1.5 (2–valued model) *A 2–valued interpretation I is called a 2–valued model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.*

Some orderings among interpretations and models will be useful:

Definition 1.1.6 (Classical ordering) *If I and J are two interpretations then we say that $I \leq J$ if $I(A) \leq J(A)$ for any ground atom A. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq I$ and $I \neq J$. An interpretation I is called least in $\mathcal{I}$ if $I \leq J$ for any other interpretation $J \in \mathcal{I}$. A model M of a program P is called minimal (resp. least) if it is minimal (resp. least) among all models of P.*

Definition 1.1.7 (Fitting ordering) *If I and J are two interpretations then we say that $I \leq_F J$ [Fitting, 1985] iff $I \subseteq J$. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called F-minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq_F I$ and $I \neq J$. An interpretation I is called F-least in $\mathcal{I}$ if $I \leq_F J$ for any interpretation $J \in \mathcal{I}$. A model M of a program P is called F-minimal (resp. F-least) if it is F-minimal (resp. F-least) among all models of P.*

Note that the classical ordering is related with the amount of true atoms, whereas the Fitting ordering is related with the amount of information, i.e. nonundefinedness.

1.2 Semantics

As argued above, a precise meaning or semantics must be associated with any logic program, in order to provide a declarative specification of it. Declarative semantics provides a mathematically precise definition of the meaning of a program, which is independent of its procedural executions, and is easy to manipulate and reason about.

In contrast, procedural semantics is usually defined as a procedural mechanism that is capable of providing answers to queries. The correctness of such a mechanism is evaluated by comparing its behaviour to the specification provided by the declarative semantics. Without the latter, the user needs an intimate knowledge of the procedural aspects in order to write correct programs.

The first attempt to provide a declarative semantics to logic programs is due to [Emden and Kowalski, 1976], and the main motivation behind their approach is based on the idea that one should minimize positive information as much as possible, limiting it to facts explicitely implied by a program, making everything else false. In other words, their semantics is based on a natural form of *"closed world assumption"* [Reiter, 1978].

Example 1.1 Consider program P :

$$\begin{array}{rcl} able\_mathematician(X) & \leftarrow & physicist(X) \\ physicist(einstein) & & \\ president(soares) & & \end{array}$$

This program has several (2–valued) models, the largest of which is the model where both Einstein and Soares are at the same time presidents, physicists and able mathematicians. This model does not correctly describe the intended meaning of P, since there is nothing in P to imply that Soares is a physicist or that Einstein is a president. In fact, the lack of such information should instead indicate that we can assume the contrary.

This knowledge is captured by the least (2–valued) model of P :

$$\{physicist(einstein), able\_mathematician(einstein), president(soares)\}$$

The existence of a unique least model for every definite program (proven in [Emden and Kowalski, 1976]), led to the definition of the so called *"least model semantics"* for definite programs. According to that semantics an atom A is true in a program P iff it belongs to the least model of P; otherwise A is false.

It turns out that this semantics does not apply to programs with default negation. For example, the program $P = \{p \leftarrow not\ q\}$ has two minimal models, namely $\{p\}$ and $\{q\}$. Thus no least model exists.

In order to define a declarative semantics for normal logic programs with negation as failure[3], [Clark, 1978] introduced the so–called *"Clark's predicate completion"*. Informally, the basic idea of completion is that in common discourse we often tend to use "if" statements when we really mean "iff" ones. For instance, we may use the following program P to describe the natural numbers:

$$\begin{array}{rcl} natural\_number(0) & & \\ natural\_number(succ(X)) & \leftarrow & natural\_number(X) \end{array}$$

This program is too weak. It does not imply that nothing but $0, 1, \ldots$ is a natural number. In fact what we have in mind regarding program P is:

$$natural\_number(X) \Leftrightarrow (X = 0 \vee (\exists Y \mid X = succ(Y) \wedge natural\_number(Y)))$$

Based on this idea Clark defined the completion of a program P, the semantics of P being determined by the 2–valued models of its completion.

However Clark's completion semantics has some serious drawbacks. One of the most important is that the completion of consistent programs may be inconsistent, thus failing to assign to those programs a meaning. For example the completion of the program $\{p \leftarrow not\ p\}$ is $\{p \Leftrightarrow not\ p\}$, which is inconsistent.

In [Fitting, 1985], the author showed that the inconsistency problem for Clark's completion can be elegantly eliminated by considering 3–valued models instead of 2–valued ones. This led to the definition of the so–called *"Fitting semantics"* for normal logic programs. In [Kunen, 1987], Kunen showed that that semantics is not recursively enumerable, and proposed a modification.

Unfortunately, the "Fitting's semantics" inherits several problems of Clark's completion, and in many cases leads to a semantics that appears to be too weak. This issue has been extensively discussed in the literature (see e.g. [Shepherdson, 1988, Przymusinski, 1989b, Gelder *et al.*, 1991]). Forthwith we illustrate some of these problems with the help of examples:

[3] In this work we adopt the designation of *"negation by default"*. Recently, this designation has been used in the literature instead of the more operational *"negation as failure"*.

Example 1.2 [4] Consider program P :

$$\begin{aligned} edge(a,b) & \\ edge(c,d) & \\ edge(d,c) & \\ reachable(a) & \\ reachable(X) & \leftarrow reachable(Y), edge(X,Y) \end{aligned}$$

that describes which vertices are reachable from a given vertice a in a graph.

Fitting semantics cannot conclude that vertices c and d are not reachable from a. Here the difficulty is caused by the existence of the symmetric rules $edge(c,d)$, and $edge(d,c)$.

Example 1.3 Consider P :

$$\begin{aligned} bird(tweety) & \\ fly(X) & \leftarrow bird(X), not\ abnormal(X) \\ abnormal(X) & \leftarrow irregular(X) \\ irregular(X) & \leftarrow abnormal(X) \end{aligned}$$

where the last two rules just state that "irregular" and "abnormal" are synonymous.

Based on the fact that nothing leads us to the conclusion that tweety is abnormal, we would expect the program to derive $not\ abnormal(tweety)$, and consequently that it flies. But Clark's completion of P is:

$$\begin{aligned} bird(X) & \Leftrightarrow X = tweety \\ fly(X) & \Leftrightarrow bird(X), not\ abnormal(X) \\ abnormal(X) & \Leftrightarrow irregular(X) \end{aligned}$$

from which it does not follow that tweety isn't abnormal.

It is worth noting that without the last two rules both Clark's and Fitting's semantics yield the expected result.

One possible explanation for such a behaviour is that the last two rules lead to a loop. This explanation is procedural in nature. But it was the idea of replacing procedural programming by declarative programming that brought about the concepts of logic programming in first place and so, as argued in [Przymusinska and Przymusinski, 1990], it seems that such a procedural explanation should be rejected.

The problems mentioned above are caused by the difficulty in representing transitive closure using completion. In [Kunen, 1988] it is formally showed that both Clark's and Fitting's semantics are not sufficiently expressive to represent transitive clousure.

In order to solve these problems some model–theoretic approaches to declarative semantics have been defined. In the beginning, such approaches did not attempt to give a meaning to every normal logic program. On the contrary, they were based on syntactic restrictions over programs, and only program complying with such restrictions were given a semantics. Examples of syntactically restricted program classes are stratified [Apt *et al.*, 1988], locally stratified [Przymusinski, 1988] and acyclic [Apt and Bezem, 1991], and examples of semantics for restricted programs are the perfect model semantics [Apt *et al.*, 1988, Przymusinski, 1988, Gelder, 1989], and the weakly perfect model semantics [Przymusinska and Przymusinski, 1988]. Here we will not review any of these approaches. For their overview, the reader is referred to e.g. [Przymusinska and Przymusinski, 1990].

[4] This example first appeared in [Gelder *et al.*, 1991].

1.2.1 Stable model semantics

In [Gelfond and Lifschitz, 1988], the authors introduce the so–called *"stable model semantics"*. This model–theoretic declarative semantics for normal programs generalizes the previously referred semantics for restricted classes of programs, in the sense that for such classes the results are the same and, moreover, for some non–restricted programs a meaning is still assigned.

The basic ideas behind the stable model semantics came for the field of nonmonotonic reasoning formalism. There, literals of the form *not* A are viewed as default literals that may or may not be assumed or, alternatively, as epistemic literals $\sim\mathcal{L}\, A$ expressing that A is not believed.

Informally, when one assumes true some set of (hypothetical) default literals, and false all the others, some consequences follow according to the semantics of definite programs [Emden and Kowalski, 1976]. If the consequences completely corroborate the hypotheses made, then they form a stable model. Formally:

Definition 1.2.1 (Gelfond–Lifschitz operator) *Let P be a normal logic program and I a 2–valued interpretation. The GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by performing the following operations:*

- *remove from P all rules which contain a default literal not A such that $A \in I$;*
- *remove from the remaining rules all default literals.*

Since $\frac{P}{I}$ is a definite program, it has a unique least model J. We define $\Gamma(I) = J$.

It turns out that fixed points of the Gelfond–Lifschitz operator Γ for a program P are always models of P. This result led to the definition of stable model semantics:

Definition 1.2.2 (Stable model semantics) *A 2–valued interpretation I of a logic program P is a stable model of P iff $\Gamma(I) = I$.*

An atom A of P is true under the stable model semantics iff A belong to all stable models of P.

One of the main advantages of stable model semantics is its close relationship with known nonmonotonic reasoning formalisms:

As proven in [Bidoit and Froidevaux, 1988], the stable models of a program P are equivalent to Reiter's default extensions [Reiter, 1980] of the default theory obtained from P by identifying each program rule:

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

with the default rule:

$$\frac{B_1, \ldots, B_n \ : \sim C_1, \ldots, \sim C_m}{H}$$

where $\sim$ denotes classical negation.

Moreover, from the results of [Gelfond, 1987], it follows directly that stable models are equivalent to Moore's autoepistemic expansions [Moore, 1985] of the theory obtained by replacing in P every default literal *not* A by $\sim\mathcal{L}\, A$ and then reinterpreting the rule connective $\leftarrow$ as material implication.

In spite of the strong relationship between logic programming and nonmonotonic reasoning, in the past these research areas were developing largely independently of one another, and the exact nature of their relationship was not closely investigated or understood.

The situation has changed significantly with the introduction of stable models, and the establishment of formal relationships between these and other nonmonotonic formalisms. In

fact, in recent years increasing and productive effort has been devoted to the study of the relationships between logic programming and several nonmonotonic reasoning formalisms. As a result, international workshops have been organized, in whose proceedings [Nerode *et al.*, 1991, Pereira and Nerode, 1993] many works and references to the theme can be found.

Such relationships turn out to be mutual beneficial. On the one hand, nonmonotonic formalisms provide elegant semantics for logic programming, specially in what regards the meaning of default negation (or negation as failure), and help one understand how logic programs can be used to formalize several types of reasoning in Artificial Intelligence. On the other hand, those formalisms benefit from the existing procedures of logic programming, and some new issues of the former are raised and solved by the latter. Moreover, relations among nonmonotonic formalisms themselves have been facilitated and established via logic programming.

1.2.2 Well–founded semantics

Despite its advantages, and of being defined for more programs than any of its predecessors, stable model semantics still has some important drawbacks:

- First, some programs have no stable models. One such program is $P = \{a \leftarrow not\ a\}$.
- Even for programs with stable models, their semantics do not always lead to the expected intended results. For example consider program P :

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ c & \leftarrow & not\ a \\ c & \leftarrow & not\ c \end{array}$$

  whose only stable model is $\{c, b\}$. Thus b and c are consequences of the stable model semantics of P. However, if one adds c to P as a lemma, the semantics of P changes, and b no longer follows. This issue is related with the property of cumulativity, and is studied in chapter 10.
- Moreover, it is easy to see that above it is impossible to derive b from P using any derivation procedure based on top–down (SL–like) rewriting techniques. This is because such a procedure, beginning with the goal $\leftarrow b$ would reach only the first two rules of P, from which b cannot be derived. This issue is related with the property of relevance, and is also studied in chapter 10.
- The computation of stable models is NP–complete [Marek and Truszczynski, 1991] even within simple classes of programs, such as propositional logic programs. This is an important drawback, specially if one is interest in a program for efficiently implementing knowledge representation and reasoning.
- Last but not least, by always insisting on 2–valued interpretations, stable model semantics often lack expressivity. This issue will be further explored in section 5.2.

The well–founded semantics was introduced in [Gelder *et al.*, 1991], and overcomes all of the above problems. This semantics is also closely related to some of the major nonmonotonic formalisms (cf. sections 5.2.1, 6.1.2, and 7.2).

Many different equivalent definitions of the well–founded semantics exist (e.g. [Przymusinski, 1989a, Przymusinski, 1989c, Bry, 1989, Przymusinska and Przymusinski, 1990, Dung, 1991, Przymusinski, 1991a, Baral and Subrahmanian, 1991, Monteiro, 1992]). Here we use the definition introduced in [Przymusinska and Przymusinski, 1990] because, in our view, it is the one

more technically related with the definition of stable models above[5]. Indeed, it consists of a natural generalization for 3–valued interpretations of the stable model semantics. In its definition the authors begin by introducing 3–valued (or partial) stable models, and then show that the F–least of those models coincides with the well–founded model as first defined in [Gelder *et al.*, 1991].

In order to formalize the notion of partial stable models, Przymusinska and Przymusinski first expand the language of programs with the additional propositional constant **u** with the property of being undefined in every interpretation. Thus they assume that every interpretation I satisfies:

$$\hat{I}(\mathbf{u}) = \hat{I}(not\ \mathbf{u}) = \frac{1}{2}$$

A non–negative program is a program whose premises are either atoms or **u**. In [Przymusinska and Przymusinski, 1990], it is proven that every non–negative program has a 3–valued least model. This led to the following generalization of the Gelfond–Lifschitz Γ–operator:

Definition 1.2.3 (Γ^*–operator) *Let P be a normal logic program, and let I be a 3–valued interpretation. The extended GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by performing the operations:*

- *remove from P all rules which contain a default literal not A such that $I(A) = 1$;*
- *replace in the remaining rules of P those default literals not A such that $I(A) = \frac{1}{2}$ by* **u**;
- *remove from the remaning rules all default literals.*

Since the resulting program is non–negative, it has a unique 3–valued least model J. We define $\Gamma^*(I) = J$.

Definition 1.2.4 (Well–founded semantics) *A 3–valued interpretation I of a logic program P is a partial stable model of P iff $\Gamma^*(I) = I$.*

The well–founded semantics of P is determined by the unique F–least partial stable model of P, and can be obtained by the (bottom–up) iteration of Γ^* starting from the empty interpretation.

[5]For a more practical introduction to the well–founded semantics the reader is referred to [Pereira *et al.*, 1991c].

Chapter 2

Extended logic programs

Recently several authors have stressed and shown the importance of including a second kind of negation $\neg$ in logic programs, for use in deductive databases, knowledge representation, and non–monotonic reasoning [Gelfond and Lifschitz, 1990, Gelfond and Lifschitz, 1992, Inoue, 1991, Kowalski, 1990, Kowalski and Sadri, 1990, Pearce and Wagner, 1990, Pereira *et al.*, 1991d, Pereira *et al.*, 1991g, Pereira *et al.*, 1992f, Pereira *et al.*, 1993d, Wagner, 1991a].

In this chapter we begin by reviewing the main motivations for introducing a second kind of negation in logic programs. Then we define an extension of the language of programs to two negations, and briefly overview the main proposed semantics for these programs.

In normal logic programs the negative information is implicit, i.e. it is not possible to explicitly state falsity, and propositions are assumed false if there is no reason to believe they are true. This is what is wanted in some cases. For instance, in the classical example of a database that explicitly states flight connections, one wants to implicitly assume that the absence of a connection in the database means that no such connection exists.

However this is a serious limitation in other cases. As argued in [Pearce and Wagner, 1990, Wagner, 1991a], explicit negative information plays an important rôle in natural discourse and commonsense reasoning. The representation of some problems in logic programming would be more natural if logic programs had some way of explicitly representing falsity. Consider for example the statement:

"Penguins do not fly"

One way of representing this statement within logic programming could be:

$$no\_fly(X) \leftarrow penguin(X)$$

or equivalently:

$$fly'(X) \leftarrow penguin(X)$$

as suggested in [Gelfond and Lifschitz, 1989].

But these representations do not capture the connection between the predicate $no\_fly(X)$ and the predication of flying. This becomes clearer if, additionally, we want to represent the statement:

"Birds fly"

Clearly this statement can be represented by

$$fly(X) \leftarrow bird(X)$$

But then, no connection whatsoever exists between the predicates $no\_fly(X)$ and $fly(X)$. Intuitively one would like to have such an obvious connection established.

The importance of these connections grows if we think of negative information for representing exceptions to rules [Kowalski, 1990]. The first statement above can be seen as an exception to the general rule that normally birds fly. In this case we really want to establish the connection between flying and not flying.

Exceptions expressed by sentences with negative conclusions are also common in legislation [Kowalski, 1989, Kowalski, 1991]. For example, consider the provisions for depriving British citizens of their citizenship:

> 40 - (1) Subject to the provisions of this section, the Secretary of State may by order deprive any British citizen to whom this subsection applies of his British citizenship if [...]
>
> (5) The Secretary of State shall not deprive a person of British citizenship under this section if [...]

Clearly, 40.1 has the logical form "P if Q" whereas 40.5 has the form "¬ P if R". Moreover, it is also clear that 40.5 is an exception to the rule of 40.1.

Above we argued for the need of having explicit negation in the head of rules. But there are also reasons that compels us to believe explicit negation is needed also in their bodies. Consider the statement[1]:

" A school bus may cross railway tracks under the condition that there is no approaching train"

It would be wrong to express this statement by the rule:

$$cross \leftarrow not\ train$$

The problem is that this rule allows the bus to cross the tracks when there is no information about either the presence or the absence of a train. The situation is different if explicit negation is used:

$$cross \leftarrow \neg train$$

Then the bus is only allowed to cross the tracks if the bus driver is sure that there is no approaching train. The difference between *not* p and $\neg p$ in a logic program is essential whenever we cannot assume that available positive information about p is complete, i.e. we cannot assume that the absence of information about p clearly denotes its falsity.

Moreover, the introduction of explicit negation in combination with the existing default negation allows for greater expressivity, and so for representing statements like:

" If the driver is not sure that a train is not approaching then he should wait"

in a natural way:

$$wait \leftarrow not\ \neg train$$

Examples of such combinations also appear in legislation. For example consider the following article from "The British Nationality Act 1981" [HMSO, 1981]:

> (2) A new–born infant who, after commencement, is found abandoned in the United Kingdom shall acquire british citizenship by section 1.2 if it is not shown that it is not the case that the person is born [...]

[1] This example is due to John McCarthy, and was published for the first time in [Gelfond and Lifschitz, 1990].

Clearly, conditions of the form "it is not shown that it is not the case that P" can be expressed naturally by $not\ \neg P$.

Another motivation for introducing explicit negation in logic programs relates to the symmetry between positive and negative information. This is of special importance when the negative information is easier to represent than the positive one. One can first represent it negatively, and then say that the positive information corresponds to its complement.

In order to make this clearer, take the following example [Gelfond and Lifschitz, 1990]:

Example 2.1 Consider a graph description based on the predicate $arc(X, Y)$, which expresses that in the graph there is an arc from vertice X to vertice Y. Now suppose that we want to determine which vertices are terminals. Clearly, this is a case where the complement information is easier to represent, i.e. it is much easier to determine which vertices are not terminal. By using explicit negation in combination with negation by default, one can then easily say that terminal vertices are those which are not nonterminal:

$$\begin{array}{rcl} \neg terminal(X) & \leftarrow & arc(X, Y) \\ terminal(X) & \leftarrow & not\ \neg terminal(X) \end{array}$$

Finally, another important motivation for extending logic programming with explicit negation is to generalize the relationships between logic programs and nonmonotonic reasoning formalisms.

As mentioned in section 1.2, such relationships, drawn for the most recent semantics of normal logic programs, have proven of extreme importance for both sides, giving them mutual benefits and clarifications. However, normal logic programs just map into narrow classes of the more general nonmonotonic formalisms. For example, simple default rules such as:

$$\frac{\sim a\ :\sim b}{c} \qquad \frac{a\ :\ b}{c} \qquad \frac{a\ :\ b}{\sim c}$$

cannot be represented by a normal logic program. Note that not even normal nor seminormal defaults rules can be represent using normal logic programs. This is so because these programs cannot represent rules with negative conclusions, and normal rules with positive conclusions have also positive justifications, which is impossible in normal programs.

Since, as shown below, extended logic programs also bear a close relationship with nonmonotonic reasoning formalisms, they improve on those of normal programs as extended programs map into broader classes of theories in nonmonotonic formalisms, and so more general relations between several of those formalisms can now be made via logic programs.

One example of such an improvement is that the introduction of explicit negation into logic programs makes it possible to represent normal and seminormal defaults within logic programming. On the one side, this provides methods for computing consequences of normal default theories. On the other, it allows for the appropriation in logic programming of work done using such theories for representing knowledge.

2.1 Language

As for normal logic programs, an atom over an alphabet $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol, and the t_is are terms. In order to extend our language with a second kind of negation, we additionally define an objective literal over $\mathcal{A}$ as being an atom A or its explicit negation $\neg A$. We also use the symbol $\neg$ to denote complementary literals in the sense of explicit negation. Thus $\neg\neg A = A$. Here, a literal is either an objective literal L or its default negation $not\ L$. We dub default literals those of the form $not\ L$.

By the extended Herbrand base of $\mathcal{A}$, we mean the set of all ground objective literals of $\mathcal{A}$. Whenever unambigous we refer to the extended Herbrand base of an alphabet, simply as Herbrand base, and denote it by $\mathcal{H}$.

An extended logic program is a finite set of rules of the form:

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

where H is an objective literal and each of the L_is is a literal. As for normal programs, if $n = 0$ we omit the arrow symbol.

By the extended Herbrand base $\mathcal{H}$ of P we mean the extended Herbrand base of the alphabet consisting of all the constants, predicate and function symbols that explicitly appear in P.

Interpretation is defined as for normal programs, but using the extended Herbrand base instead.

Whenever unambigous, we refer to extended logic programs simply as logic programs or programs. As in normal programs, a set of rules stands for all its ground instances.

In the sequel we refer to some special forms of programs:

Definition 2.1.1 (Canonical program) *An extended logic program P is a canonical program iff for every rule in P*

$$H \leftarrow Body$$

if $L \in Body$ then not $\neg L \in Body$, where L is any objective literal.

Definition 2.1.2 (Semantics kernel) *An extended logic program P is a semantics kernel iff every rule in P is of the form:*

$$H \leftarrow not\ L_1, \ldots, not\ L_n \qquad (n \geq 0)$$

2.2 Semantics

The first semantics defined for extended logic programs was the so–called *"answer–sets semantics"* [Gelfond and Lifschitz, 1990]. There the authors defined for the first time the language of logic programs with two kinds of negation – default negation *not* and what they called classical negation $\neg$.

The answer–sets semantics is a generalization of the stable model semantics for the language of extended programs. Roughly, an answer–set of an extended program P is a stable model of the normal program obtained from P by replacing objective literals of the form $\neg L$ by new atoms, say $\neg\_L$.

Definition 2.2.1 (The Γ–operator) *Let P be an extended logic program and I a 2–valued interpretation. The GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by:*

- *first denoting every objective literal in $\mathcal{H}$ of the form $\neg A$ by a new atom, say $\neg\_A$;*
- *replacing in both P and I, these objective literals by their new denotation;*
- *then performing the following operations:*
  - *removing from P all rules which contain a default literal not A such that $A \in I$;*
  - *removing from the remaning rules all default literals.*

Since $\frac{P}{I}$ is a definite program it has a unique least model J.

If J contains a pair of complementary atoms, say A and $\neg\_A$, then $\Gamma(I) = \mathcal{H}$.

Otherwise, let J' be the interpretation obtained from J by replacing the newly introduced atoms $\neg\_A$ by $\neg A$. We define $\Gamma(I) = J'$.

Definition 2.2.2 (Answer–set semantics) *A 2–valued interpretation I of an extended logic program P is an answer–set model of P iff $\Gamma(I) = I$.*

An objective literal L of P is true under the answer–set semantics iff L belongs to all answer–sets of P; L is false iff $\neg L$ is true; otherwise L is unknown.

In [Gelfond and Lifschitz, 1990], the authors showed that the answer–sets of an extended program P are equivalent to Reiter's default extensions of the default theory obtained from P by identifying each program rule:

$$H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m, not\ D_1, \ldots, not\ D_k, not\ \neg E_1, \ldots, not\ \neg E_j$$

with the default rule:

$$\frac{B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_m\ :\sim D_1, \ldots, \sim D_k, E_1, \ldots, E_j}{H'}$$

where $H' = H$ if H is an atom, or $H' =\sim L$ if $H = \neg L$.

Another semantics generalizing stable models for the class of extended programs is the e–answer–set semantics of [Kowalski and Sadri, 1990]. There, the authors claim that explicitly negated atoms in extended programs play the rôle of exceptions. Thus they impose a preference of negative over positive objective literals.

The e–answer–set semantics is obtainable from the answer–set semantics after a suitable program transformation. For the sake of simplicity, here we do not give the formal definition of e–answer–sets, but instead show its behaviour in an example:

Example 2.2 Consider program P :

$$\begin{aligned} fly(X) &\leftarrow bird(X) \\ \neg fly(X) &\leftarrow penguin(X) \\ bird(X) &\leftarrow penguin(X) \\ penguin(tweety) & \end{aligned}$$

This program allows for both the conclusions $fly(tweety)$ and $\neg fly(tweety)$. Thus its only answer–set is $\mathcal{H}$.

In e–answer–set semantics, since conclusions of the form $\neg L$ are preferred over those of the form L, $\neg fly(tweety)$ overrides the conclusion $fly(tweety)$, and thus

$$\{penguin(tweety), bird(tweety), \neg fly(tweety)\}$$

is an e–answer–set of P.

The rationale for this overriding is that the second rule is an exception to the first one.

In [Przymusinski, 1990a], the author argues that the technique used in answer–sets for generalizing stable models is quite general. Based on that he defines a semantics which generalizes the well–founded semantics for the class of extended programs[2], as follows:

[2] In the sequel we refer to this semantics as *"well–founded semantics with pseudo negation"*. The justification for this name can be found in section 5.1.4.

Definition 2.2.3 (Well–founded semantics with pseudo negation) *A 3–valued interpretation I is a partial stable model of an extended logic program P iff I' is a partial stable model of the normal program P', where I' and P' are obtained respectively from I and P, by replacing every objective literal of the form $\neg A$ by a new atom, say $\neg\_A$.*

The well–founded semantics with pseudo negation of P is determined by the unique F–least partial stable model of P.

Based on the notions of vivid logic [Levesque, 1986] and strong negation [Nelson, 1949], [Wagner, 1991a] presents an alternative definition of the answer–set semantics. There, the author claims that the $\neg$–negation of extended logic programs is not classical negation but rather Nelson's strong negation.

In fact, consider the following program P :

$$\begin{array}{lcl} b & \leftarrow & a \\ b & \leftarrow & \neg a \end{array}$$

If *real* classical negation were used then b would be a consequence of P, because for classical negation $a \vee \neg a$ is a tautology. However, in neither of the above mentioned semantics b follows from P.

In order to introduce *real* classical negation into logic programs, in [Przymusinski, 1991b] the author defines the "stationary semantics with classical negation". This semantics is a generalization of the well–founded semantics, and is capable of deriving b in P. For brevity we do not present here its formal definition. However, the definition can be found in section 5.1, where we compare it with our *WFSX*.

Unlike normal logic programs, none of the semantics of extended programs is defined for every program, i.e. some programs are contradictory. While for some programs this seems reasonable (e.g. a program containing contradictory facts, say $P = \{a \leftarrow, \quad \neg a \leftarrow\}$), for others this can be too strong:

Example 2.3 Let P :

$$\begin{array}{lcl} \neg p & \leftarrow & not\ q \\ p & & \end{array}$$

In all the above semantics this program is not assigned a meaning. Roughly, this is because q has no rules, and thus *not* q must be true. So, by the first rule, $\neg p$ must also be true, and since there is a fact p in P, a contradiction appears.

However, if we see default literals as hypotheses that may or may not be assumed (viz. in [Dung, 1991]), this contradiction seems strange since it relies on the assumption of *not* q.

Motivated by this [Dung and Ruamviboonsuk, 1991] presented a semantics generalizing "well–founded semantics with pseudo negation" which, in order to assign a meaning to more programs, do not assume hypotheses (default literals) that lead to a contradiction[3]. For instance, the semantics of P above does not assume *not* q, and is $\{p\}$.

Other researchers have defined paraconsistent semantics for contradictory programs e.g. [Sakama, 1992, Wagner, 1993]. This is not our nor Dung's concern. On the contrary, we wish to remove contradiction whenever it rests on withdrawable assumptions.

[3]At the same conference, we presented a paper [Pereira *et al.*, 1991a] exploring similar ideas. The details of that independent work are not presented in this overview but are expounded at length in chapter 8.

Part II

A New Semantics for Extended Logic Programs

Chapter 3

Why a new semantics for extended programs?

The overview above showed several semantics exist for extended logic programs. In our view none correctly captures the meaning of extended programs. This is why we think a new semantics for extended programs is required. Let's look at their shortcomings:

The answer–set semantics [Gelfond and Lifschitz, 1990], being based on the stable model semantics of normal program [Gelfond and Lifschitz, 1988], suffers at least from the same structural and computational problems of the latter. We briefly recall some of those problems (as pointed out in section 1.2.2):

- Some noncontradictory programs have no answer–sets, e.g. $P = \{a \leftarrow not\ a\}$.
- Even for programs with answer–sets, their semantics does not always render the expected intended results. In particular (cf. the example of page 15), the addition of lemmas changes the semantics of the program (this is related with the property of cumulativity studied in chapter 10)
- Derivation procedures for answer–sets cannot be based on top–down (SL–like) rewriting techniques (this is related with the property of relevance also studied in chapter 10). For example consider the program:

  $$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ c & \leftarrow & not\ a \\ \neg c & & \end{array}$$

  whose only answer–set is $\{\neg c, a\}$.

  Though a is a consequence of this program, a does not follow from the rules "below"[1] a, which in this case are the first two.

  Indeed, the program containing only the first two rules has two answer–sets: $\{a\}$ and $\{b\}$. Thus neither a nor b are true in this program.
- The computation of answer–sets is NP–complete, even within simple classes of programs such as propositional logic programs. Moreover, for non–propositional programs, in general it is impossible to compute answer–sets by finite approximations (as shown in section 7.5).
- By always insisting on 2–valued interpretations, answer–set semantics often lacks expressibility. This issue is further explored in section 5.2.

[1] For the formalization of what we mean by "below" see section 10.1.2.

The e–answer–sets semantics of [Kowalski and Sadri, 1990] also inherits the same problems of stable models. Moreover, we think that explicitly negated atoms do not always represent exceptions. For example consider the statements:

- *Animals do not fly.*
- *Birds fly.*
- *Birds are animals.*
- *Ozzy is a bird.*

Here the second statement (with a positive conclusion) is an exception to the first (with a negative conclusion). Of course, in this case we can represent these statements using a predicate $no\_fly(X)$, thereby making the first rule have a positive conclusion and the second a negative one. However this technique cannot be used if, additionally, we want to represent:

- *Penguins do not fly.*
- *Penguins are birds.*
- *Tweety is a penguin.*

If one represents all the statements using predicate $fly(X)$:

$$\begin{aligned} \neg fly(X) &\leftarrow animal(X) \\ fly(X) &\leftarrow bird(X) \\ \neg fly(X) &\leftarrow penguin(X) \\ \\ animal(X) &\leftarrow bird(X) \\ bird(X) &\leftarrow penguin(X) \\ bird(ozzy) & \\ penguin(tweety) & \end{aligned}$$

then the only e–answer–set contains $\neg fly(ozzy)$ because it is an animal, which is not intuitively correct since $ozzy$ is a bird and so it should fly.

If one represents the statements using predicate $no\_fly(X)$, then the only e–answer–set contains $\neg no\_fly(tweety)$ because it is a bird, which again is not intuitively correct since $tweety$ is a penguin and so it should not fly.

In our view, a declarative semantics for extended programs should not impose any preference between positive and explicit negative information. Their treatment should be symmetric. It is up to the programmer to, for each specific case, write his program in such a way that the desired preferences are made. In section 6.8.1 we show an example of how to write a program imposing preferences of exceptions over rules. The systematization of a representation method for rules and exceptions using extended logic programs is, however, beyond the scope of this work. For that the reader is referred to [Pereira *et al.*, 1991g, Pereira *et al.*, 1993b, Aparício, 1993].

The semantics of [Przymusinski, 1990a] based on the well–founded semantics does not suffer from the problems of answer–sets. Moreover it does not impose any preference of negative atoms over positive ones.

Unfortunately, because [Przymusinski, 1990a] uses the same technique for adding explicit negation to well–founded semantics as answer–sets for stable models semantics, important properties which relate both negations, obeyed by answer–sets, are lost:

Example 3.1 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ \neg a & & \end{array}$$

If $\neg a$ were simply to be considered as a new atom symbol, say, $\neg\_a$, and well–founded semantics were used to define the meaning of P (as suggested in [Przymusinski, 1990a]), the result would be

$$\{\neg a, not\ \neg b\}$$

so that $\neg a$ is true and a is undefined. This clearly severs the connection between both negations.

In our view, $\neg a$ is an explicit declaration of the falsity of a. Thus, it can always be assumed that a is false by default, i.e. *not* a should also be true.

Example 3.2 Consider a program containing the rules:

$$\begin{array}{rcl} tryBus & \leftarrow & not\ driversSrike \\ \neg driversStrike & & \end{array}$$

advising to plan a trip by bus if it can be assumed the bus drivers are not on strike, and stating bus drivers are not on strike. No matter what the rest of the program is (assuming it is noncontradictory on the whole), it is clear that it should be assumed the bus drivers are not on strike, and of course the trip should be planned by bus.

Intuitively, $\neg driversStrike$ implies *not* $driversSrike$.

In order to relate both negations in extended logic programs, we introduce the *"coherence principle"*:

> *"Let L be an objective literal of an extended logic program P.*
> *If $\neg L$ belongs to the semantics of P then not L must also belong to the semantics of P."*

and argue that every semantics should comply with this principle[2].

Answer–set semantics complies with coherence. Simply note that, for noncontradictory programs, if $\neg L$ is in an answer–set then L is not in that answer–set and so, answer–sets being two valued, *not* L is true.

The semantics presented in [Dung and Ruamviboonsuk, 1991], being a generalization of the semantics of [Przymusinski, 1990a], does not also comply with coherence.

The issue, dealtwith by [Dung and Ruamviboonsuk, 1991], of assigning meaning to more programs by unassuming default literals leading to contradiction is, in our view, an important one. However, we think this should be done on the basis of a coherent semantics, and that its result should also comply with coherence. In chapter 8, we show how to deal with contradictory programs, when the contradiction is brought about by default literals. There we present a more sceptical semantics (in the spirit of [Dung and Ruamviboonsuk, 1991]) that avoids contradiction and complies with coherence. Then we show this same semantics can be obtained by using instead a contradiction removal process that transforms programs considered contradictory. The advantages of using the latter instead of the former approach are presented in section 8.4.

Finally, also the "well–founded semantics with classical negation" of [Przymusinski, 1991b] does not capture the intuitive meaning of extended programs. This happens because of its very first motivation, i.e. the introduction of *real* classical negation.

[2]More arguments in favour of the coherence principle can be found spread along this work.

Consider again the program:

$$\begin{array}{lcl} b & \leftarrow & a \\ b & \leftarrow & \neg a \end{array}$$

whose well–founded semantics with classical negation entails b.

We recall that the intended meaning of $\neg L$ is that L is explicitly false or, in other words, L is known to be false. With this reading of explicit negation, the rules of the program state that if a is known to be true then b is known to be true, and if a is known to be false then b is known to be true. Given that the knowledge about literals is not always complete, i.e. it might happen that a is neither known to be false nor true, the formula $a \vee \neg a$ is not a tautology. So the law of excluded middle does not apply, and b does not follow from these statements.

Our stance is that if the law of excluded middle is desired of some atom A then so much should be explicitly stated by adding the disjunctive rule $A \vee \neg A$. This expresses that the knowledge about A is complete, i.e. A is known to be either true or false. We have yet to enlarge our language for rules to accomodate such expressiveness.

In section 5.2.2–"Why $\neg$ should not be classical negation", we further explore this view of explicit negation, by comparing extended programs with logics of knowledge and belief. There, we argue that explicit negation $\neg L$ should have the reading "L is known to be false", and justify that classical negation in extended program corresponds to "it is false that L is known to be true" or "L is not known to be true" or, conflating knowledge with truth, as classical logic does, "L is not true"; whereas *not* L reads "L is not believed".

Another property not obeyed by classical negation in logic program is supportedness. Roughly, a semantics complies with supportedness if, for every program P, an objective literal L is true only if there is an identifiable rule for L whose body is true[3]. Clearly, this property closely relates to the use of logic as a programming language. One does not expect an objective literal to be true unless some identifiable rule with true body concludes it; in other words, every true objective literal must be solely supported on other definitely true objective literals or on the truth of default literals. Such is the nature of epistemic truth or knowledge. Ontological truth is concerned with truth in the world, not with the epistemically justifiable knowledge an agent may hold. Thus, in the ontological stance $L \vee \sim L$ is true regardless of whether any of the cases is supported.

[3]For a formal definition of this property see section 5.1.3.

Chapter 4

WFSX – A well founded semantics for extended logic programs

In this chapter we present a new semantics for normal logic programs (i.e. with negation by default) extended with explicit negation, that subsumes the well founded semantics [Gelder *et al.*, 1991] of normal programs.

Parts of this chapter appear in [Pereira and Alferes, 1992] and in [Pereira *et al.*, 1992d].

4.1 Interpretations and models

We begin by providing definitions of interpretation and model for programs extended with explicit negation.

Definition 4.1.1 (Interpretation) *An interpretation I of a language Lang is any set*

$$T \cup not\ F\ ^{1}$$

where T and F are disjoint subsets of objective literals over the Herbrand base, and:

$$if\ \neg L \in T\ then\ L \in F\ \textit{(Coherence Principle)}^{2}.$$

The set T contains all ground objective literals true *in I, the set F contains all ground objective literals* false *in I. The truth value of the remaining objective literals is* undefined.

Notice how the two types of negation become linked via coherence: for any objective L, if $\neg L \in I$ then $not\ L \in I$. Other semantics introducing a second negation in WFS do not relate the two negation in this way (cf. chapter 5 on comparisons).

This definition of interpretation not only guarantees that every interpretation complies with coherence but also with noncontradiction.

Proposition 4.1.1 (Noncontradiction condition) *If $I = T \cup not\ F$ is an interpretation of a program P then there is no pair of objective literals A, $\neg A$ of P such that $A \in T$ and $\neg A \in T$.*

Proof: (by contradiction) Consider that $I = T \cup not\ F$ is such that $A \in T$ and $\neg A \in T$. By the coherence condition $A \in F$ and $\neg A \in F$. So I is not an interpretation because T and F are not disjoint. $\diamondsuit$

[1] Where $not\ \{a_1, \ldots, a_n\}$ stands for $\{not\ a_1, \ldots, not\ a_n\}$.

[2] For any literal L, if L is explicitly false L must be false. Note that the complementary condition "if $L \in T$ then $\neg L \in F$" is implicit.

Example 4.1 $\{a, \neg a, \neg b\}$ is not an interpretation because a and $\neg a$ belong to it (contradiction) and also because *not* b does not belong to it although $\neg b$ does (incoherence).

An interpretation I can be read intuitively in the following way:

- An atom A is *true* (resp. *explicitly false*) in I iff $A \in I$ (resp. $\neg A \in I$).
- A positive (resp. negative) objective literal A (resp. $\neg A$) is *false* in I iff *not* $A \in I$ (resp. *not* $\neg A \in I$).
- An atom A is *undefined* in I otherwise.

As in [Przymusinska and Przymusinski, 1990], an interpretation can be equivalently viewed as a function $I : \mathcal{H} \rightarrow V$ where $\mathcal{H}$ is the set of all objective literals in the language and $V = \left\{0, \frac{1}{2}, 1\right\}$.

Proposition 4.1.2 *Any interpretation $I = T \cup not\ F$ can be equivalently viewed as a function $I : \mathcal{H} \rightarrow V$ where $V = \left\{0, \frac{1}{2}, 1\right\}$, defined by:*

$$\begin{array}{rll} I(A) = 0 & \textit{if} & \textit{not } A \in I; \\ I(A) = 1 & \textit{if} & A \in I; \\ (A) = \frac{1}{2} & & \textit{otherwise.} \end{array}$$

Based on this function we can define a truth valuation of formulae.

Definition 4.1.2 (Truth valuation) *If I is an interpretation, the truth valuation $\hat{I}$ corresponding to I is a function $\hat{I} : C \rightarrow V$ where C is the set of all formulae of the language, recursively defined as follows:*

- *if L is an objective literal then $\hat{I}(L) = I(L)$.*
- *if $S = not\ L$ is a default literal then $\hat{I}(not\ L) = 1 - I(L)$.*
- *if S and V are formulae then $\hat{I}((S, V)) = min(\hat{I}(S), \hat{I}(V))$.*
- *if L is an objective literal and S is a formula then:*

$$\hat{I}(L \leftarrow S) = \begin{cases} 1 & \textit{if } \hat{I}(S) \leq \hat{I}(L) \textit{ or } \hat{I}(\neg L) = 1 \textit{ and } \hat{I}(S) \neq 1 \\ 0 & \textit{otherwise} \end{cases}$$

The only additional condition with respect to WFS (cf. definition 1.1.3 above), $\hat{I}(\neg L) = 1$ and $\hat{I}(S) \neq 1$, does not affect the valuation of formulae without $\neg$. Its purpose is to allow a conclusion c to be independently false when the premises are undefined for some rule, on condition that $\neg c$ holds. This allows, in particular, explicit negation $\neg$ to override with false the undefinedness of conclusions of rules with undefined bodies.

Definition 4.1.3 (Model) *An interpretation I is called a model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.*

Example 4.2 The models of the program:

$$\begin{array}{lllll} \neg b & b & \leftarrow & a & \qquad c \leftarrow not\ \neg c \\ & a & \leftarrow & not\ a, not\ c & \qquad \neg c \leftarrow not\ c \end{array}$$

are:

$$
\begin{aligned}
M_1 &= \{\neg b, not\ b\} \\
M_2 &= \{\neg b, not\ b, c, not\ \neg c\} \\
M_3 &= \{\neg b, not\ b, c, not\ \neg c, not\ a\} \\
M_4 &= \{\neg b, not\ b, not\ c, \neg c\} \\
M_5 &= \{\neg b, not\ b, \neg a, not\ a\} \\
M_6 &= \{\neg b, not\ b, \neg a, not\ a, c, not\ \neg c\} \\
M_7 &= \{\neg b, not\ b, not\ \neg a\} \\
M_8 &= \{\neg b, not\ b, c, not\ \neg c, not\ \neg a\} \\
M_9 &= \{\neg b, not\ b, c, not\ \neg c, not\ a, not\ \neg a\} \\
M_{10} &= \{\neg b, not\ b, not\ c, \neg c, not\ \neg a\}
\end{aligned}
$$

Only M_3, M_6, and M_9 are models in the usual sense (i.e. classical models in the sense of definition 1.1.4).

- M_1, M_2, M_4, M_7, M_8, and M_{10} are not classical models, because in all of them the body of the rule $b \leftarrow a$ is undefined and the head is false, i.e. the truth value of the head is smaller than that of the body.
- M_5 is not a classical model since in it the truth value of the head (false) of rule $a \leftarrow not\ a, not\ c$ is smaller than that of the head (undefined).

4.2 The definition of *WFSX*

Next we introduce the notion of stability in models, and using it we define the *WFSX* semantics.

As in [Przymusinska and Przymusinski, 1990], in order to define the semantics, we expand the language by adding to it the proposition $\mathbf{u}$ such that every interpretation I satisfies $I(\mathbf{u}) = \frac{1}{2}$. By a non–negative program we also mean a program whose premises are either objective literals or $\mathbf{u}$.

We extend with an additional operation the P modulo I transformation of [Przymusinska and Przymusinski, 1990], itself an extension of the Gelfond-Lifschitz modulo transformation [Gelfond and Lifschitz, 1988].

Definition 4.2.1 ($\frac{P}{I}$ transformation) *Let P be an extended logic program and let I be an interpretation. $\frac{P}{I}$, P modulo I, is the program obtained from P by performing in the sequence the following four operations:*

- *Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$.*
- *Remove from P all rules containing an objective literal L such that $\neg L \in I$.*
- *Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$.*
- *Replace all the remaining default literals by proposition $\mathbf{u}$.*

Note that the new operation, the second one, is not applicable to non–extended programs, and is only needed by some extended programs. It is required by the coherence principle, as illustrated below in this section.

The resulting program $\frac{P}{I}$ is by definition non–negative.

Definition 4.2.2 (Least operator) *We define $least(P)$, where P is a non–negative program, as the set of literals $T \cup not\ F$ obtained as follows:*

- *Let P' be the non–negative program obtained by replacing in P every negative objective literal $\neg L$ by a new atomic symbol, say $'\neg\_L'$.*
- *Let $T' \cup not\ F'$ be the least 3–valued model of P' (cf. definition 1.1.6).*
- *$T \cup not\ F$ is obtained from $T' \cup not\ F'$ by reversing the replacements above.*

The least 3–valued model of a non–negative program can be defined as the least fixpoint of the following generalization of the Van Emden–Kowalski least model operator Ψ for definite logic programs:

Definition 4.2.3 (Ψ^* operator) *Suppose that P is a non–negative program, I is an interpretation of P and A and the A_i are all ground atoms. Then $\Psi^*(I)$ is a set of atoms defined as follows:*

- $\Psi^*(I)(A) = 1$ *iff there is a rule* $A \leftarrow A_1, \ldots, A_n$ *in P such that $I(A_i) = 1$ for all $i \leq n$.*
- $\Psi^*(I)(A) = 0$ *iff for every rule* $A \leftarrow A_1, \ldots, A_n$ *there is an $i \leq n$ such that $I(A_i) = 0$.*
- $\Psi^*(I)(A) = 1/2$, *otherwise.*

Theorem 4.2.1 (3–valued least model) *The 3–valued least model of a non–negative program is:*

$$\Psi^{*\uparrow\omega}(not\ \mathcal{H})$$

The generalization of the Van Emden–Kowalski theorem set forth in [Przymusinska and Przymusinski, 1990] is also valid for extended logic of programs.

Theorem 4.2.2 *least(P) uniquely exists for every non–negative program P.*

Proof: Since P' is a non–negative program without explicit negation its least 3–valued model M exists and is unique (by theorem 6.24 of [Przymusinska and Przymusinski, 1990] page 357). The theorem follows since *least*(P) is univocally obtained from M. $\diamondsuit$

Note that *least*(P) isn't always an interpretation in the sense of definition 4.1.1. Conditions about noncontradiction and coherence may be violated.

Example 4.3 Consider the non–negative program P :

$$\begin{array}{rcl@{\qquad}rcl} a & \leftarrow & & \neg a & \leftarrow & \neg b \\ \neg b & \leftarrow & & b & \leftarrow & \mathbf{u} \end{array}$$

where $least(P) = \{a, \neg a, \neg b\}$. This set is not an interpretation (cf. example 4.1). Noncontradiction and coherence are violated.

Example 4.4 Consider the program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ b \\ \neg a & & \end{array}$$

and the interpretation $I = \{\neg a, not\ a\}$.

$$\frac{P}{I} = \begin{array}{rcl} a & \leftarrow & \mathbf{u} \\ b & \leftarrow & \mathbf{u} \\ \neg a & & \end{array}$$

So, $least\left(\frac{P}{I}\right) = \{\neg a\}$.

Although noncontradictory this set of literals violates coherence.

To impose coherence, when contradiction is not present, we define a partial operator that transforms any noncontradictory set of literals into an interpretation.

Definition 4.2.4 (The *Coh* operator) *Let $QI = QT \cup not\ QF$ be a set of literals such that QT does not contain any pair of objective literals A, $\neg A$. $Coh(QI)$ is the interpretation $T \cup not\ F$ such that*

$$T = QT \text{ and } F = QF \cup \{\neg L \mid L \in T\}.$$

The Coh operator is not defined for contradictory sets of literals.

The result of Coh applied to $least\left(\frac{P}{I}\right)$ is always an interpretation. The noncontradiction and coherence conditions are guaranteed by definition. T and F are disjoint because QT and QF are disjoint and none of the objective literals added to F are in T since T is noncontradictory.

Now we generalize the Γ^* operator of [Przymusinska and Przymusinski, 1990].

Definition 4.2.5 (The Φ operator) *Let P be a logic program, I an interpretation, and $J = least\left(\frac{P}{I}\right)$.*

If $Coh(J)$ exists then $\Phi_P(I) = Coh(J)$. Otherwise $\Phi_P(I)$ is not defined.

Definition 4.2.6 (*WFSX*, PSM and WFM) *An interpretation I of an extended logic program P is called an Partial Stable Model (PSM) of P iff*

$$\Phi_P(I) = I.$$

The F-least Partial Stable Model is called the Well Founded Model(WFM).

The WFSX *semantics of P is determined by the set of all PSMs of P.*

It is easy to see that some programs may have no *WFSX* semantics.

Example 4.5 The program $P = \{a \leftarrow,\ \ \neg a \leftarrow\}$ has no semantics.

Definition 4.2.7 (Contradictory program) *An extended logic program P is contradictory iff it has no semantics, i.e. there exists no interpretation I such that $\Phi_P(I) = I$.*

Theorem 4.3.5 below expresses an alternative, more illustrative definition of contradictory program. The issue of handling contradictory programs is further discussed in chapter 8.

Example 4.6 Consider again the program of example 3.1.

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ \neg a & & \end{array}$$

Now $\{\neg a, not\ \neg b\}$ is no longer a PSM as in [Przymusinski, 1990a] (where $\neg a$ and $\neg b$ are simply considered new atoms), because it is not an interpretation, and thus Φ does not apply to it.

Its only PSM, and consequently its WFM, is:

$$I = \{\neg a, b, not\ a, not\ \neg b\}.$$

$$\frac{P}{I} = \begin{array}{rcl} b & \leftarrow \\ \neg a & \leftarrow \end{array}$$

Indeed, its least model is I, $Coh(I) = I$, and $\Phi_P(I) = I$.

Remark 4.2.1 *According to [Przymusinski, 1990a], the above program has two PSMs:*

$$\{\neg a, not\ \neg b\} \quad \text{and} \quad \{\neg a, b, not\ a, not\ \neg b\}$$

only the second being coherent. It is not enough though to throw out those of his models not complying with coherence. Although that's true for this example, example 4.7 shows that's not the general case.

Example 4.7 Consider program P:

$$\begin{array}{lllll} c & \leftarrow & not\ b & \quad a & \leftarrow\ not\ a \\ b & \leftarrow & not\ a & \quad \neg b & \leftarrow \end{array}$$

Applying the semantics to P we have the model:

$$PSM = \{\neg b, c, not\ b, not\ \neg c, not\ \neg a\}.$$

Indeed:

$$\frac{P}{PSM} = \begin{array}{lllll} c & \leftarrow & & \quad a & \leftarrow\ \mathbf{u} \\ b & \leftarrow & \mathbf{u} & \quad \neg b & \leftarrow \end{array}$$

its least model is $\{c, \neg b, not\ \neg c, not\ \neg a\}$, and consequently

$$\Phi_P(PSM) = PSM^3.$$

By simply considering $\neg b$ as a new atom (as suggested in [Przymusinski, 1990a]) this non–extended program would have a single PSM, $\{\neg b\}$, which is not a coherent interpretation.

It is also interesting to notice in this example that PSM is not a model in the classical sense because for the second rule of P the value of the head ($PSM(b) = 0$) is smaller than the value of the body ($PSM(not\ a) = \frac{1}{2}$).

The intuitive idea is that the truth of $\neg b$ (or the independent falsity of b) overrides any rule for b with undefined body, so that $not\ b$ becomes true (and b false), rather than undefined. This is important to allow if we consider the existence of the fact $\neg b$ in the program instrumental in specifying the falsity of b in it. In chapter 5 section 5.2 this issue is further discussed.

Even though PSMs are not models in the classical sense, they are models as defined above in this chapter (definition 4.1.3).

Theorem 4.2.3 (PSMs are models) *Every PSM of a program P is a model of P.*

Proof:(by contradiction) Let I be a PSM and not a model of P. By definition of model:

$$\hat{I}(L \leftarrow B) \neq 1$$

only if

$$\hat{I}(L) < \hat{I}(B) \text{ and } \hat{I}(B) = 1, \quad \text{or} \quad \hat{I}(L) < \hat{I}(B) \text{ and } \hat{I}(\neg L) \neq 1.$$

If the first disjunct holds, then since $\hat{I}(B) = 1$ and I is a PSM, $L \in I$ (i.e. $\hat{I}(L) = 1$), so the disjunct cannot hold.

If the second disjunct holds, then

$$\text{either } \hat{I}(B) = 1 \quad \text{or} \quad \hat{I}(B) = \frac{1}{2}.$$

[3]Note how the truth of $\neg b$ compels the truth of $not\ b$ via the Coh operator.

The first case is impossible, as just shown. If $\hat{I}(B) = \frac{1}{2}$ then:

$$least\left(\frac{P}{I}\right)(L) = \frac{1}{2}$$

and since $\hat{I}(\neg L) \neq 1$:

$$Coh\left(least\left(\frac{P}{I}\right)\right)(L) = \frac{1}{2}.$$

As I is a PSM, $\hat{I}(L) = \frac{1}{2} = \hat{I}(B)$, so the disjunct cannot hold. $\diamondsuit$

Example 4.8 Consider example 4.2. The only PSMs of that program correspond exactly to models M_7, M_9 and M_{10}.

We now come back to the question of the need for the extra operation introduced in the modulo transformation.

Example 4.9 Consider program P :

$$\begin{array}{rcl@{\quad}rcl} c & \leftarrow & a & a & \leftarrow & b \\ \neg a & \leftarrow & & b & \leftarrow & not\ b \end{array}$$

Its only PSM is $I = \{\neg a, not\ a, not\ c, not\ \neg b, not\ \neg c\}$. In fact,

$$\frac{P}{I} = \begin{array}{rcl@{\quad}rcl} & & & a & \leftarrow & b \\ \neg a & \leftarrow & & b & \leftarrow & \mathbf{u} \end{array}$$

$$least\left(\frac{P}{I}\right) = \{\neg a, not\ c, not\ \neg b, not\ \neg c\}$$

and consequently $\Phi(I) = I$.

If the new operation for the modulo transformation were absent, $\frac{P}{I}$ would contain the rule $c \leftarrow a$, and c would be undefined rather than false. This would go against the coherence principle, since $\neg a$ entails $not\ a$, and as the only rule for c has a in the body, it should also entail $not\ c$. The rôle of the new operation is to ensure the propagation of false as a consequence of any $not\ L$ implied by a $\neg L$ through coherence.

Consider now a similar program P', in the canonical (cf. definition 2.1.1) form:

$$\begin{array}{rcl@{\quad}rcl} c & \leftarrow & a, not\ \neg a & a & \leftarrow & b, not\ \neg b \\ \neg a & \leftarrow & & b & \leftarrow & not\ b \end{array}$$

Its only PSM is again $I = \{\neg a, not\ a, not\ c, not\ \neg b, not\ \neg c\}$.

$$\frac{P'}{I} = \begin{array}{rcl@{\quad}rcl} & & & a & \leftarrow & b \\ \neg a & \leftarrow & & b & \leftarrow & \mathbf{u} \end{array}$$

Because of the canonical form the new operation of the modulo transformation is irrelevant. Even without it the rule $c \leftarrow a, not\ \neg a$ is removed by applying the first operation, given that $\neg a \in I$ and that $not\ \neg a$ is part of its body.

In general, for programs in the canonical form the second operation of the modulo operator is no longer required.

Theorem 4.2.4 (Compact version of $\frac{P}{I}$) *Let P be an canonical extended logic program, and I an interpretation. Then $\frac{P}{I}$ can be equivalently defined as the program obtained from P by performing in sequence the three operations:*

- *Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$.*
- *Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$.*
- *Replace all the remaining default literals by proposition* **u**.

Proof: Trivial, given the definitions of canonical program, of interpretation, and of $\frac{P}{I}$. $\diamondsuit$

4.3 Existence of the semantics

In the above definition of the semantics (definition 4.2.6) we define the WFM as the F–least PSM. This is possible because:

Theorem 4.3.1 (Existence of the semantics) *For noncontradictory programs there always exists a unique F–least PSM. Moreover a literal L belongs to every PSM of a noncontradictory program P iff L belong to the F–least PSM of P.*

Proof: The proof follows directly from theorem 4.3.2 below. $\diamondsuit$

Theorem 4.3.2 (Monotonicity of Φ) *Let P be a noncontradictory program. Then the operator Φ_P is monotonic wrt set inclusion, i.e. $A \subseteq B \Rightarrow \Phi_P(A) \subseteq \Phi_P(B)$ for any interpretations A and B.*

Proof: Since $\Phi_P(I) = Coh\Big(least\Big(\frac{P}{I}\Big)\Big)$ we prove this theorem by proving two lemmas, concerning respectively the monotonicity of Coh and that of $least\Big(\frac{P}{I}\Big)$.

Lemma 4.3.3 *Consider a program P and let $I = T_I \cup not\ F_I$ and $J = T_J \cup not\ F_J$ be two interpretations of P such that $I \subseteq J$. $Coh(I) \subseteq Coh(J)$ holds.*

Proof: $Coh(I) \subseteq Coh(J)$ is equivalent, by definition of Coh, to

$$T_I \cup not\ (F_I \cup \{\neg L \mid L \in T_I\}) \subseteq T_J \cup not\ (F_J \cup \{\neg L \mid L \in T_J\})$$

since $T_I \subseteq T_J$ by hypothesis, the above is true if:

$$F_I \cup \{\neg L \mid L \in T_I\} \subseteq F_J \cup \{\neg L \mid L \in T_I\} \cup \{\neg L \mid L \in T_J - T_I\}$$

which is equivalent to

$$F_I \subseteq F_J \cup \{\neg L \mid L \in T_J - T_I\}$$

which holds because, by hypothesis, $F_I \subseteq F_J$. $\diamondsuit$

Lemma 4.3.4 *Consider a program P and let $I = T_I \cup not\ F_I$ and $J = T_J \cup not\ F_J$ be two interpretations of P such that $I \subseteq J$.*

$$least\Big(\frac{P}{I}\Big) \subseteq least\Big(\frac{P}{J}\Big)\ holds.$$

Proof: In [Przymusinska and Przymusinski, 1990] this is proven considering the modulo transformation without the second rule. Since this rule does not introduce new undefined literals, it does not affect the monotonicity of the operator. $\diamondsuit$

Now it is easy to complete the proof of the theorem. By lemma 4.3.4:

$$A \subseteq B \ \Rightarrow\ least\Big(\frac{P}{A}\Big) \subseteq least\Big(\frac{P}{B}\Big)$$

and by lemma 4.3.3:

$$least\left(\frac{P}{A}\right) \subseteq least\left(\frac{P}{B}\right) \;\Rightarrow\; Coh\left(least\left(\frac{P}{A}\right)\right) \subseteq Coh\left(least\left(\frac{P}{B}\right)\right)$$

for a noncontradictory program P. $\diamondsuit$

Definition 4.3.1 (Iterative construction of the WFM) *In order to obtain a constructive bottom–up definition of the WFM of a given noncontradictory program P, we define the following transfinite sequence $\{I_\alpha\}$ of interpretations of P:*

$$\begin{array}{rcl} I_0 & = & \{\} \\ I_{\alpha+1} & = & \Phi_P(I_\alpha) \\ I_\delta & = & \bigcup\{I_\alpha \mid \alpha < \delta\} \quad \textit{for a limit ordinal } \delta \end{array}$$

By theorem 4.3.2, and according to the Knaster–Tarski theorem [Tarski, 1955], there must exist a smallest ordinal λ such that I_λ is a fixpoint of Φ_P, and $WFM = I_\lambda$.

Top–down procedures computing this semantics can be easily obtained by adapting existing procedures for WFS of programs without explicit negation, such as [Pereira *et al.*, 1991e, Pereira *et al.*, 1992e], as follows: replace every literal of the form $\neg A$ by a new literal, say A'; include two new rules *"not A rewrites to A'"* and *"not A' rewrites to A"*. If A and A' are both derivable then the program is contradictory.

The constructive bottom–up definition requires one to know *a priori* if the given program is contradictory. This requirement is not needed if we consider the following theorem.

Theorem 4.3.5 *A program P is contradictory iff in the sequence of I_α there exists a λ such that $\Phi_P(I_\lambda)$ is not defined, i.e. $least\left(\frac{P}{I_\lambda}\right)$ has a pair of objective literals A, $\neg A$.*

Proof: The theorem is equivalent to: P is noncontradictory iff in the sequence of I_α there exists no λ such that $\Phi_P(I_\lambda)$ is not defined.

If P is noncontradictory then Φ_P is monotonic, and so no such λ exists. If there is no such λ then there exists an I and a smallest α such that $I = \Phi_P^{\uparrow\alpha}(\{\})$, and I is a fixpoint of Φ_P. Thus, a fixpoint of Φ_P exists, and so P is noncontradictory. $\diamondsuit$

In order to (bottom–up) compute the WFM of a program P start by building the above sequence. If at some step Φ_P is not applicable then end the iteration and conclude that P is contradictory. Otherwise, iterate until the least fixpoint of Φ_P, which is the WFM of P.

Example 4.10 Consider program P:

$$\begin{array}{rcl} a & \leftarrow & not\ a \\ \neg a & \leftarrow & \end{array}$$

Let us build the sequence:

$$\begin{array}{rcl} I_0 & = & \{\} \\ I_1 & = & Coh\left(least\left(\frac{P}{\{\}}\right)\right) = Coh(least(\{a \leftarrow \mathbf{u}, \quad \neg a \leftarrow\})) \\ & = & Coh(\{\neg a\}) = \{\neg a, not\ a\} \\ I_2 & = & Coh\left(least\left(\frac{P}{\{\neg a, not\ a\}}\right)\right) = Coh(least(\{a \leftarrow, \quad \neg a \leftarrow\})) \\ & = & Coh(\{a, \neg a\}) \end{array}$$

which is not defined. So P is contradictory.

Example 4.11 Consider program P of example 4.6. The sequence is:

$$\begin{array}{rcl} I_0 & = & \{\} \\ I_1 & = & Coh\Big(least\Big(\frac{P}{\{\}}\Big)\Big) = Coh(least(\{a \leftarrow \mathbf{u}, \quad b \leftarrow \mathbf{u}, \quad \neg a \leftarrow\})) \\ & = & Coh(\{\neg a, not\ \neg b\}) = \{\neg a, not\ a, not\ \neg b\} \\ I_2 & = & Coh\Big(least\Big(\frac{P}{\{\neg a, not\ a, not\ \neg b\}}\Big)\Big) \\ & = & Coh(least(\{a \leftarrow \mathbf{u}, \quad b \leftarrow, \quad \neg a \leftarrow\})) \\ & = & Coh(\{b, \neg a, not\ \neg b\}) = \{b, \neg a, not\ a, not\ \neg b\} = I_3 \end{array}$$

and thus the WFM of P is $\{b, \neg a, not\ a, not\ \neg b\}$.

It is worth noting that this semantics is a generalization of the well–founded semantics to programs with explicit negation.

Theorem 4.3.6 (Generalization of the well–founded semantics) *For programs without explicit negation* WFSX *coincides with well–founded semantics.*

Proof: As noted before, the modulo transformation coincides with the one defined for stationary semantics for the case of non–extended programs. Furthermore, the additional conditions imposed on interpretations are void for those programs and, finally, the Coh operator reduces to identity. $\diamondsuit$

Chapter 5

WFSX, LP semantics with two negations, and autoepistemic logics

In recent years increasing and productive effort has been devoted to the study of the relationships between logic programming and several nonmonotonic reasoning formalisms[1]. Such relationships are mutual beneficial. On the one hand, nonmonotonic formalisms provide elegant semantics for logic programming, specially in what regards the meaning of default negation (or negation as failure), and help one understand how logic programs are used to formalize several types of problems in Artificial Intelligence. On the other hand, those formalisms benefit from the existing procedures of logic programming, and some new issues of the former are raised and solved by the latter. Moreover, relations among nonmonotonic formalisms have been facilitated and established via logic programming.

For normal logic programs, their relationship with default theories [Reiter, 1980] was first proposed in [Bidoit and Froidevaux, 1987]. In [Eshghi and Kowalski, 1989] default negation of normal programs was first formalized as abduction, and in [Dung, 1991] the idea was further explored in order to capture stable models [Gelfond and Lifschitz, 1988] and the well–founded semantics [Gelder *et al.*, 1991] of normal programs.

The idea of viewing logic programs as autoepistemic theories first appeared in [Gelfond, 1987] where the author proposed to view every negated literal *not* L of logic programs as $\sim\mathcal{L}\, L$,[2] i.e. *not* L has the epistemic reading: *"there is no reason to believe in* L*"*. In [Bonatti, 1992], different transformations between default negation literals and belief literals are studied, in order to show how different logic programming semantics can be obtained from autoepistemic logics.

However, except for our previous work, incorporated here, no such general comparisons exist for extended logic programming. The establishment of relationships between nonmonotonic formalisms and extended logic programs improve on those for normal programs since extended programs map into broader classes of theories in nonmonotonic formalisms, and so more general relations between several of those formalisms can now be made via logic programs. Moreover, the relationships also provide a clearer meaning of the $\neg$–negation and its relation to default negation in extended logic programming.

In this and the next chapters we explore the relationship between extended logic programs and several nonmonotonic formalisms: autoepistemic logic, default theory, abduction, and belief revision.

[1] As a result, international workshops have been organized, in whose proceedings [Nerode *et al.*, 1991, Pereira and Nerode, 1993] many additional references can be found.

[2] In the sequel we refer to this transformation, between default negation literals and belief literals, as the Gelfond transformation.

The first part of this chapter is devoted to contrasting and characterizing a variety of semantics for extended logic programs, including *WFSX*, in what concerns their use and meaning of ¬-negation, and its relation to both classical negation and the default negation, *not* , of normal programs.

For this purpose we define a parametrizeable schema to encompass and characterize a diversity of proposed semantics for extended logic programs, where the parameters are two: one the axioms $AX_{\neg}$ defining ¬-negation; another the minimality conditions not_{cond}, defining *not* -negation.

By adjusting these parameters in the schema we can then specify several semantics involving two kinds of negation [Gelfond and Lifschitz, 1990, Pereira and Alferes, 1992, Przymusinski, 1990a, Przymusinski, 1991b, Wagner, 1991a], including *WFSX*. Other semantics, dealing with contradiction removal [Pereira *et al.*, 1992b, Dung and Ruamviboonsuk, 1991, Pereira *et al.*, 1991a, Sakama, 1992], are not directly addressed by the schema, though the issue is touched upon in section 5.2.4. The issue of contradiction in extended logic programming is studied in length in chapter 8.

In the second part of this chapter, and based on the similarities between the parametrizable schema and the definitions of autoepistemic logics, we proceed to examine the relationship between them and extended logic programs.

In the above mentioned comparative study, concerning the use and meaning of ¬–negation in different semantics, no epistemic meaning is assigned to each of the uses of ¬. By relating extended logic programs to autoepistemic logics such a meaning is extracted for some cases. In particular, we show that $\neg L$ in *WFSX* can be read as *"L is known to be false"*. Other semantics give different readings to ¬, e.g. in the stationary semantics with classical negation of [Przymusinski, 1991b] $\neg L$ has the epistemic reading: *"L is not known to be true"*.

These results also clarify the use of logic programs for representing knowledge and belief.

5.1 Generic semantics for programs with two kinds of negation

The structure of this section is as follows: we begin with preliminary definitions and subsection 5.1.2 presents the parametrizeable schema; next we present properties important for the study of extended logic program semantics, and show for various $AX_{\neg}$ whether or not the resulting semantics complies with such properties; afterwards, in subsection 5.1.4, we reconstruct the plurality of semantics for extended logic programs in the schema by specifying, for each, their set $AX_{\neg}$ and their condition not_{cond}; finally we briefly address the issue of introducing disjunction in extended logic programs.

Parts of this section appear in [Alferes and Pereira, 1992].

5.1.1 Preliminaries

In the sequel, we translate every extended logic program P into a set of general clauses $\neg\_P$, which we dub *clausal logic program.* A set of general clauses is, as usual, a set of clauses:

$$L_1 \vee \ldots \vee L_n$$

where each L_i is either an atom A or its *classical negation* $\sim A$. Here, by classical negation we mean the negation of classical logic. Just as it was important to distinguish between classical negation and negation by default in order to develop the relationship between normal logic programming and nonmonotonic reasoning, here it is equally important to distinguish between explicit negation ¬ and *real* classical negation $\sim$, specially because our concern is to better characterize the former.

The models and interpretations of clausal logic programs are simply the classical models and interpretations of sets of general clauses.

Propositions of the form *not_A* (the translation in the clausal logic program $\neg\_P$ for *not A* in P) are called *default* ones, all other propositions being *objective* ones.

5.1.2 Stationary and stable semantics for programs with two kinds of negation

Within this section we present the above mentioned parametrizeable schema. We begin by defining two generic semantics for normal logic programs extended with an extra kind of negation: one extending the stationary semantics [Przymusinski, 1990b, Przymusinski, 1991b] for normal programs (itself equivalent to well founded semantics [Gelder *et al.*, 1991]); another extending the stable model semantics [Gelfond and Lifschitz, 1988]. We dub each of these semantics generic because they assume little about the extra kind of negation introduced. The meaning of the negation by default is however completely determined in each of the two generic semantics (both stationary and stable models) that we present.

Subsequently we generalize the schema in order to parametrize it w.r.t. negation by default as well.

Stationary semantics for programs with two kinds of negation

Here we redefine the stationary semantics of [Przymusinski, 1991b] in order to parametrize it with a generic second type of negation, in addition to negation by default. We start by defining stationary expansion of normal programs as in [Przymusinski, 1991b].

Definition 5.1.1 (Minimal models) *A minimal model of a theory (or set of general clauses) T is a model M of T with the property that there is no smaller model N of T which coincides with M on default propositions.*

If a formula F is true in all minimal models of T then we write:

$$T \models\!\!\!\models_{CIRC} F$$

and say that F is minimally entailed by T.

This amounts to McCarthy's Parallel Circumscription [McCarthy, 1980]:

$$CIRC(T; \mathcal{O}; \mathcal{D})$$

of theory T in which objective propositions $\mathcal{O}$ are minimized and default propositions $\mathcal{D}$ are fixed.

Definition 5.1.2 (Stationary expansion of normal programs) *A stationary expansion of a normal program P is any consistent theory P^* which satisfies the fixed point condition:*

$$P^* = \neg\_P \cup \Big\{not\_A \mid P^* \models\!\!\!\models_{CIRC} {\sim}A\Big\} \cup \Big\{{\sim}not\_A \mid P^* \models\!\!\!\models_{CIRC} A\Big\}$$

where A is any arbitrary atom, $\neg\_P$ is the program obtained from P by replacing every literal of the form not L by not_L.

Note that $\neg\_P$ and P^* are always sets of Horn clauses.

Example 5.1 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ a \\ b & \leftarrow & not\ a, c \\ d & \leftarrow & not\ b \end{array}$$

whose clausal program is $\neg\_P$:

$$\begin{array}{c} a \vee \sim not\_a \\ b \vee \sim not\_a \vee \sim c \\ d \vee \sim not\_b \end{array}$$

The only expansion of P is

$$P^* = \neg\_P \cup \{not\_b, not\_c, \sim not\_d\}$$

In fact the minimal models of P^* are (for clarity throughout the examples we exhibit all literals, both positive and negative):

$$\begin{array}{llllllll} \{ & not\_a, & not\_b, & not\_c, & \sim not\_d, & a, & \sim b, & \sim c, \quad d \quad \} \\ \{ & \sim not\_a, & not\_b, & not\_c, & \sim not\_d, & \sim a, & \sim b, & \sim c, \quad d \quad \} \end{array}$$

As P^* entails $\sim b$, $\sim c$, and d, it must contain $\{not\_b, not\_c, \sim not\_d\}$ and no more default literals.

As proven in [Przymusinski, 1991b], the least stationary expansion of a normal program gives its well–founded semantics (via a definition of meaning similar to definition 5.1.4), and now we wish to extend WFS with explicit negation to obtain, among others, *WFSX*.

In order to extend this definition to logic programs with a generic second kind of negation $\neg$, we additionally transform any such negated literals into new atoms too:

Definition 5.1.3 (Clausal program $\neg\_P$ of P) *The clausal program $\neg\_P$ of an extended logic program P is the clausal set of Horn clauses obtained by first denoting every literal in $\mathcal{H}$ of the form:*

$$\begin{array}{rll} \neg A & by\ a\ new\ atom & \neg\_A \\ not\ A & by\ a\ new\ atom & not\_A \\ not\ \neg A & by\ a\ new\ atom & not\_\neg\_A \end{array}$$

then replacing in P such literals by their new denotation and, finally, reinterpreting the rule connective $\leftarrow$ as material implication, expressed by $\Rightarrow$.

Example 5.2 Let $P = \{a \leftarrow \neg b\}$. The clausal program $\neg\_P$ is:

$$\neg\_P = \{\neg\_b \Rightarrow a\}$$

or equivalently:

$$\neg\_P = \{a \vee \sim\neg\_b\}.$$

The models of an extended program are determined by the models of its clausal program expansions via an inverse transformation:

Definition 5.1.4 (Meaning of a clausal program P^*) *The meaning of a clausal program expansion P^* is the union of the sets of all atoms:*

$$\begin{array}{rll} A & such\ that & P^* \models A \\ \neg A & such\ that & P^* \models \neg\_A \\ not\ A & such\ that & P^* \models not\_A \\ not\ \neg A & such\ that & P^* \models not\_\neg\_A \end{array}$$

where $P^* \models L$ means that literal L belongs to all (classical) models of (the set of general clauses) P^*.

Note that negative literals do not translate over.

In order to specify the second kind of negation one introduces in $\neg\_P$ the axioms $AX_\neg$ defining it. For example, if we want the second negation to be classical negation we must add to $\neg\_P$ the set of clauses

$$\{\neg\_A \Leftrightarrow \sim A \mid A \in \mathcal{H}\}$$

where $\Leftrightarrow$ denotes material equivalence, and is used as shorthand for both clauses $\neg\_A \Rightarrow \sim A$ and $\sim A \Rightarrow \neg\_A$. In this case, the semantics of P is the same whether or not the first part of the transformation to $\neg\_P$ takes place.

We want this generic semantics to be an extension of stationary semantics. So we must guarantee that the semantics of a program without any occurence of $\neg$-negation is the same as for stationary semantics, whatever kind of $\neg$-negation axioms are used and defined in the generic schema. To that end, we must first minimize by circumscription the atoms in the language of P, and only afterwards do we minimize the bar–ed atoms.

Definition 5.1.5 ($M \bar{\leq} N$) *Let M and N be two models of a program $\neg\_P$ and M_{pos} (resp. N_{pos}) be the subset of M (resp. N) obtained by deleting from it all literals of the form $\neg\_L$.*

We say that $M \bar{\leq} N$ iff:

$$M_{pos} \subseteq N_{pos} \vee (M_{pos} = N_{pos} \wedge M \subseteq N).$$

This definition is similar to the classical one plus a condition to the effect that, say, model $M_1 = \{\neg\_a\}$ is smaller than model $M_2 = \{a\}$.

Minimal models are now defined as in 5.1.1 but with this new $\bar{\leq}$ relation. The equivalence between minimality and circumscription is made through the ordered predicate circumscription $CIRC(T; \mathcal{O}; \mathcal{D})$ of the theory T, in which objective propositions $\mathcal{O}$ are minimized, but minimizing first propositions not of the form $\neg\_A$, and only afterwards the latter, and where default propositions $\mathcal{D}$ are fixed parameters.

The definition of stationary expansion of an extended programs is then a generalization of definition 5.1.2, parametrized by the set of axioms $AX_\neg$ defining $\neg\_A$, plus this new notion of ordered minimality.

Definition 5.1.6 (Stationary $AX_\neg$ expansions) *A stationary expansion of an $AX_\neg$ extended program P is any consistent theory P^* which satisfies the following fixed point condition:*

$$P^* = \neg\_P \cup AX_\neg \cup \left\{not\_L \mid P^* \models_{\overline{CIRC}} \sim L\right\} \cup \left\{\sim not\_L \mid P^* \models_{\overline{CIRC}} L\right\}$$

where L is any arbitrary objective proposition, and $AX_\neg$ is the set of axioms for $\neg$-negation in P.

A stationary expansion P^* of a program P is obtained by adding to the corresponding clausal program $\neg\_P$ the axioms defining $\neg$–negation, and the negations by default not_L of those and only those literals L which are false in all minimal models of P^*. The meaning of negation by default is that, in any stationary expansion P^*, not_L holds if and only if P^* minimally entails $\sim L$. Note that the definition of $AX_\neg$ can influence, by reducing the number of models, whether $\sim L$ is in all minimal models of P^*.

It is known (cf. [Lifschitz, 1985, Etherington *et al.*, 1985, Gelfond *et al.*, 1989]) that for any positive proposition A of any theory T, the above definition of $\models_{\overline{CIRC}}$ implies:

$$T \models_{\overline{CIRC}} A \equiv T \models A$$

Thus, and directly from definition 5.1.6:

Proposition 5.1.1 *A consistent theory P^* is a stationary expansion of an $AX_\neg$ extended program P iff:*

- *P^* is obtained by augmenting $\neg\_P \cup AX_\neg$ with some default propositions not_A and $\sim not\_A$ where A is an objective proposition;*
- *P^* satisfies the conditions:*

$$\begin{array}{lllll} P^* \models & not\_A & \equiv & P^* \models_{CIRC} & \sim A \\ P^* \models \sim & not\_A & \equiv & P^* \models & A \end{array} \qquad \text{and}$$

for any objective proposition A.

Example 5.3 Consider program P :

$$\begin{array}{lll} p & \leftarrow & a \\ p & \leftarrow & \neg a \\ q & \leftarrow & not\ p \end{array}$$

where $\neg$ in P is classical negation, i.e.

$$AX_\neg = \{\neg\_a \Leftrightarrow \sim a,\ \neg\_p \Leftrightarrow \sim p,\ \neg\_q \Leftrightarrow \sim q\}.$$

The clausal program of P is:

$$\begin{array}{lll} p & \vee & \sim a \\ p & \vee & \sim \neg\_a \\ q & \vee & \sim not\_p \end{array}$$

The only stationary expansion of P is:

$$P_1^* = \neg\_P \cup AX_\neg \cup \{\sim not\_p, not\_\neg\_p, not\_q, \sim not\_\neg\_q, not\_a, \sim not\_\neg\_a\}$$

In fact, the only minimal model of P_1^* is:

$$\{\sim not\_p, not\_\neg\_p, not\_q, \sim not\_\neg\_q, not\_a, \sim not\_\neg\_a, \\ p, \sim \neg\_p, \sim q, \neg\_q, \sim a, \neg\_a \}$$

and the conditions of proposition 5.1.1 hold.

Note how the $\bar{\leq}$ relation prefers this model to other models that would be minimal if the usual $\leq$ were to be enforced. For example, the classically minimal model:

$$\{\sim not\_p, not\_\neg\_p, not\_q, \sim not\_\neg\_q, not\_a, \sim not\_\neg\_a, \\ p, \sim \neg\_p, q, \sim \neg\_q, \sim a, \neg\_a \}$$

is not minimal when the $\bar{\leq}$ relation is considered.

If $\neg$ in P is defined by :

$$AX_\neg = \{\neg\_a \Rightarrow \sim a,\ \neg\_p \Rightarrow \sim p,\ \neg\_q \Rightarrow \sim q\}$$

i.e. $\neg$ in P is a *strong* negation in the sense that it implies classical negation in $\neg\_P$, then the only stationary expansion of P is:

$$P_2^* = \neg\_P \cup AX_\neg \cup \{not\_p, not\_\neg\_p, \sim not\_q, not\_\neg\_q, not\_a, not\_\neg\_a\}$$

In fact, the only minimal model of P_2^* is:

$$\{not\_p, not\_\neg\_p, \sim not\_q, not\_\neg\_q, not\_a, not\_\neg\_a, \sim p, \sim \neg\_p, q, \sim \neg\_q, \sim a, \sim \neg\_a\}$$

and the conditions of proposition 5.1.1 hold.

We now define the semantics of a program based on its stationary expansions relative to some $AX_{\neg}$.

Definition 5.1.7 (Stationary $AX_{\neg}$ semantics) *A stationary $AX_{\neg}$ model of a program P is the meaning of P^*, where P^* is a stationary $AX_{\neg}$ expansion of P.*

The stationary $AX_{\neg}$ semantics of an extended program P is the set of all stationary $AX_{\neg}$ models of P.

If $S = \{M_k \mid k \in K\}$ is the semantics of P, then the intended meaning of P is:

$$M = \bigcap_{k \in K} M_k.$$

Example 5.4 The meaning of the program of example 5.3 is:

$$\{p, \neg q, \neg a, not\ q, not\ a, not\ \neg p\}$$

if we use classical negation, and:

$$\{q, not\ p, not\ \neg p, not\ \neg q, not\ a, not\ \neg a\}$$

if we use strong negation.

Example 5.5 Consider P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ \neg a & & \end{array}$$

where $\neg$ is a weak form of negation determined by:

$$AX_{\neg} = \{\sim A \Rightarrow \neg\_A \mid A \in \mathcal{H}\}.$$

The only stationary expansion of P is:

$$P^* = \neg\_P \cup AX_{\neg} \cup \{\sim not\_a, \sim not\_\neg\_a, not\_b, \sim not\_\neg\_b\}$$

determining thus the meaning of P as

$$M = \{a, \neg a, not\ b, \neg b\}.$$

The fact that both a and $\neg a$ belong to M is not a problem since the weak form of negation allows that. Note that $\sim A \Rightarrow \neg\_A$ is equivalent to $A \vee \neg\_A$, and allows models with both A and $\neg\_A$. Literal $\neg b$ also appears in M forced by the weak negation.

Now we state in what sense this semantics is a generalization of stationary semantics:

Proposition 5.1.2 (Generalization of stationary semantics) *Let P be a (non–extended) normal logic program, and let $AX_{\neg}$ be such that no clause of the form*

$$A_1 \vee \ldots \vee A_n \text{ where } \{A_1, \ldots, A_n\} \subseteq \mathcal{H}$$

is a logical consequence of it.

M is a stationary $AX_{\neg}$ model of P iff M (modulo the $\neg$-literals) is a stationary model of P.

The reader can check that all sets of axioms $AX_{\neg}$ used in the sequel satisfy the restriction imposed in the proposition. This restriction on the form of $AX_{\neg}$ is meant to avoid unusual definitions of $\neg$-negation where positive literals are just a consequence of the axioms independently from the program. For instance:

Example 5.6 Let $P = \{a \leftarrow b\}$, and

$$AX_{\neg} = \{a \vee \sim\neg\_b, \neg\_b\}.$$

P has a stationary $AX_{\neg}$ model

$$\{a, not\ \neg a, not\ b, \neg b\}$$

which is not a stationary model of P. Note however that a is in the model because it is a logical consequence of $AX_{\neg}$ irrespective of the program.

The parametrizeable schema

Stable Models Semantics [Gelfond and Lifschitz, 1988] has a one-to-one correspondence with stable expansions [Moore, 1985], and the latter can be obtained simply by replacing $\models\!\!\!\!\sim_{CIRC}$ by $\models\!\!\!\!\sim_{CWA}$ in the definition of stationary expansion of normal programs, where CWA denotes Reiter's closed world assumption [Reiter, 1978], as shown in [Przymusinski, 1991b].

As with the stationary semantics of extended programs, a generic definition of stable semantics for extended programs can also be obtained, with $P^* \models\!\!\!\!\sim_{CWA} \sim L$ as the condition for adding negation by default.

So, in general a new parameter in the schema is desirable in order to specify how default negation is to be added to an expansion.

Definition 5.1.8 ($\langle AX_\neg, not_{cond}\rangle$ expansion) *A $\langle AX_\neg, not_{cond}\rangle$ expansion of an extended program P is any consistent theory P^* which satisfies the following fixed point condition:*

$$P^* = \neg\_P \cup AX_\neg \cup \{not\_L \mid not_{cond}(L)\} \cup \{\sim not\_L \mid P^* \models L\}$$

where L is any arbitrary objective proposition.

The definition of a generic semantics is similar to that of stationary semantics.

Definition 5.1.9 ($\langle AX_\neg, not_{cond}\rangle$ semantics) *A $\langle AX_\neg, not_{cond}\rangle$ model of a program P is the meaning of P^*, where P^* is a $\langle AX_\neg, not_{cond}\rangle$ expansion of P.*

The semantics of a program P is the set of all $\langle AX_\neg, not_{cond}\rangle$ models of P. The intended meaning of P is the intersection of all models of P.

We define Stable $AX_\neg$ Semantics as the generic semantics where:

$$not_{cond}(L) = P^* \models\!\!\!\!\sim_{CWA} \sim L.$$

With this definition, propositions 5.1.1 and 5.1.2 are also valid for stable models.

5.1.3 Properties required of $\neg$

In this section we present some of the properties of extended logic programs and show for some $AX_\neg$ whether or not the resulting semantics comply with such properties. Here we examine the cases of:

- *classical negation* i.e. $AX_\neg = \{\ \neg\_A \Leftrightarrow \sim A \mid A \in \mathcal{H}\}$
- *strong negation* i.e. $AX_\neg = \{\ \neg\_A \Rightarrow \sim A \mid A \in \mathcal{H}\}$
- *weak negation* i.e. $AX_\neg = \{\ \sim A \Rightarrow \neg\_A \mid A \in \mathcal{H}\}$
- *pseudo negation* i.e. $AX_\neg = \{\}$.

for both the stationary and stable semantics generic schemes. In section 5.1.4 we redefine $WFSX$, introducing *explicit negation*, by imposing:

$$AX_\neg = \{\} \quad \text{and} \quad not_{cond}(L) = P^* \models\!\!\!\!\sim_{CIRC} \sim L \vee P^* \models \neg\_L$$

Alternatively, we can define *WFSX* via stationary $AX_\neg$ semantics with:

- *explicit negation* i.e. $AX_\neg = \{\neg\_A \Rightarrow not\_A \mid A \in \mathcal{H}\}$

We concentrate next only on properties concerning the $\neg$-negation. For a comparative study of semantics also concerning negation by default see section 10.2.

Property 5.1.1 (Intrinsic consistency) *A semantics is intrinsically consistent iff, for any program P, if M is a stationary (resp. stable) model of P then for no atom $A \in \mathcal{H}$:*

$$\{A, \neg A\} \subseteq M.$$

In other words, a semantics is intrinsically consistent if there is no need for testing for consistency within the final (stationary or stable) models of a program.

Example 5.7 Let P be:

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ \neg a & \leftarrow & not\ b \end{array}$$

where $\neg$ is weak negation.

The only stationary expansion of P is:

$$P^* = \neg\_P \cup \{\sim A \Rightarrow \neg\_A \mid A \in \mathcal{H}\} \cup \{not\_b, not\_\neg\_b\}.$$

The only minimal model of P^* is:

$$\{a, \sim not\_a, \neg\_a, \sim not\_\neg\_a, \sim b, \sim\neg\_b, not\_b, not\_\neg\_b\}$$

and is consistent.

However the meaning of P^* :

$$\{a, \neg a, not\_b, not\_\neg\_b\}$$

is inconsistent.

As shown with the previous example, semantics with weak negation might not be intrinsically consistent. The same happens with semantics with pseudo negation.

Semantics with classical or strong negation are intrinsically consistent because, by the very definition of $AX_\neg$, for every atom $A \in \mathcal{H}$,

$$\sim A \vee \sim\neg\_A \in P^*,$$

for every expansion P^* of any program P, and thus no model of P^* has A and $\neg\_A$. So the meaning of P^* can never contain both A and $\neg A$.

Property 5.1.2 (Coherence) *A semantics is coherent iff, for any program P and objective literal L, whenever M is a stationary (resp. stable) model of P :*

- *if $\neg L \in M$ then not $L \in M$*[3].

As argued above, this property plays an important rôle if we consider the second kind of negation instrumental for specifying the falsity of literals. In that case coherence can be read as:

if A is declared false then it must be assumed false by default.

It turns out that, for both stationary and stable semantics, coherence is equivalent to consistency:

Theorem 5.1.1 *A stationary (or stable) semantics is coherent iff it is consistent.*

[3] If $L = \neg A$, this reads as $\neg\neg A = A \in M$ then *not* $\neg A \in M$.

Proof: In appendix. ♢

Property 5.1.3 (Supportedness) *A semantics is necessarily supportive iff, for any program P, whenever M is a stationary (resp. stable) model of P then, for every objective literal L, if $L \in M$ there exists in P at least one identifiable rule of the form:*

$$L \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

such that:

$$\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\} \subseteq M.$$

Since for any program P :

$$\neg\_P \cup \left\{not\_L \mid P^* \models_{CIRC} \sim L\right\}$$

is a Horn clause program, a stationary (or a stable) semantics such that $AX_{\neg}$ does not contain any clause with positive propositions is necessarily supportive. Thus, semantics with pseudo or strong negation are necessarily supportive.

Semantics that introduce in $AX_{\neg}$ such clauses might not be necessarily supportive. For example, if $\neg$ is classical negation necessary supportedness does not hold:

Example 5.8 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & b \\ \neg a & & \end{array}$$

The only stationary $\{\neg\_A \Leftrightarrow \sim A\}$ model is:

$$M = \{not\ a, \neg a, not\ b, \neg b\}.$$

As $\neg b \in M$, and there is no rule for $\neg b$, the semantics is not necessarily supportive.

This property closely relates to the use of logic as a programming language. One does not expect objective literals to be true unless rules stating their truth condition are introduced; in other words, except for default propositions, no implicit information should be expected. We argue that if one wants the result of the previous program one should write:

$$\begin{array}{rcl} \neg b & \leftarrow & \neg a \\ \neg a & & \end{array}$$

or, if disjunction is introduced:

$$\begin{array}{rcl} a & \leftarrow & b \\ \neg a & & \\ b & \vee & \neg b \end{array}$$

5.1.4 Fixing the set $AX_{\neg}$ and the condition $not_{cond}(L)$

In this section we reconstruct some semantics for extended programs simply by specifying the set $AX_{\neg}$ and the condition $not_{cond}(L)$ w.r.t. the generic semantics defined above. We contribute this way for a better understanding of what type of second negation each of those semantics uses, what are the main differences among them, and how they compare to *WFSX*.

We begin by reconstructing answer-sets semantics [Gelfond and Lifschitz, 1990] for programs with consistent answer-sets (equivalent to the semantics of [Wagner, 1991a]).

Theorem 5.1.2 (Answer–sets semantics) *An interpretation M is an answer–set of a program P iff M is a stable*

$$AX_{\neg} = \{\neg\_A \Rightarrow \sim A \mid A \in \mathcal{H}\}$$

model of P (modulo the syntactic representation of models[4]).

Proof: Since:

- stable models correspond to stable expansions for normal logic programs, and
- answer sets are the consistent stable models of the normal program obtained by considering every objective literal of the form $\neg L$ as a new atom $\neg\_L$, i.e. consistent stable $\{\}$ models of P,

for proving this theorem it is enough to prove that:

1. All stable $\{\neg\_A \Rightarrow \sim A \mid A \in \mathcal{H}\}$ expansions are consistent
2. Consistent stable $\{\}$ models are equivalent to stable $\{\neg\_A \Rightarrow \sim A \mid A \in \mathcal{H}\}$ models.

The first point is clear given that, as shown in section 5.1.3, stable semantics with strong negation are always consistent.

If P^* is a consistent stable $\{\}$ expansion, then for every objective proposition $\neg\_A$:

$$P^* \models \neg\_A \overset{\text{by consistency}}{\Rightarrow} P^* \not\models A \overset{\text{by CWA}}{\Leftrightarrow} P^* \models_{\overline{CWA}} \sim A$$

Thus, formulae of the form:

$$\neg\_A \Rightarrow \sim A$$

are theorems in all consistent stable $\{\}$ models.

So, by adding them to expansions the results remain unchanged, i.e. point 2 holds. $\diamondsuit$

This theorem leads to the conclusion that answer-sets semantics extends stable models semantics with strong negation. Thus, from the results of section 5.1.3, we conclude that answer-sets semantics is consistent, coherent and supportive.

Note that if instead of strong negation one uses pseudo negation and a test for consistency in the final models, the result would be the same. However, we think that the formalization as in theorem 5.1.2 is more accurate because the consistency there is intrinsic and dealt within the fixpoint condition, with no need for meta–level constraints, and the properties exhibited are those of strong negation and not of pseudo negation. For example, coherence and intrinsic consistency (properties of strong negation but not of pseudo negation) are obeyed by answer-sets semantics.

One semantics extending well founded semantics with $\neg$-negation is presented in [Przymusinski, 1990a], and reviewed in section 2.2 above. It claims that the method used in [Gelfond and Lifschitz, 1990] can be applied to semantics other than stable models, and so that method is used to define the proposed semantics. It happens that the meaning of $\neg$ is not the same as for answer-sets, in the sense that different $AX_{\neg}$s are used:

Theorem 5.1.3 (WFS plus $\neg$ as in [Przymusinski, 1990a]) *An interpretation M is an extended stable model of a program P iff M is a consistent stationary $AX_{\neg} = \{\}$ model of P.*

[4]Recall that in the definition of answer–sets, default literals are not included in models. By "modulo the syntactic representation of models" we mean removing all default literals in models according to this definition.

Proof: Trivial, given that for normal logic programs WFS corresponds to stationary models, and WFS plus $\neg$ as in [Przymusinski, 1990a] is just the WFS of the normal program obtained by considering literals of the form $\neg L$ simply as new atoms $\neg\_L$. $\diamondsuit$

Note the need for testing consistency in stationary models of the semantics so that L and $\neg L$ are related in the end. As seen in section 5.1.3, this semantics does not comply with coherence, which imposes a permanent relationship between L and $\neg L$ in the computation of models.

Next we reconstruct the stationary semantics with classical negation presented in [Przymusinski, 1991b]. This semantics is originally defined similarly to the generic definition above, but where $AX_{\neg}$ is absent and literals of the form $\neg A$ and $not\ \neg A$ are just transformed to $\sim A$ and $\sim not\ A$, respectively. From this similarity the reconstruction follows easily:

Theorem 5.1.4 (Stationary semantics with classical negation) *An interpretation M is a stationary model (in the sense of [Przymusinski, 1991b]) of a program P iff M is a stationary*

$$AX_{\neg} = \{\neg\_A \Leftrightarrow \sim A \mid A \in \mathcal{H}\}$$

model of P.

Proof: In appendix. $\diamondsuit$

From the results of section 5.1.3 we conclude that this semantics does not comply with supportedness. Nevertheless, this semantics is the only one reconstructed here that introduces *real* classical negation into normal logic programs. We argue that, comparing it with semantics with strong negation, this is not a big advantage since, once disjunction is added to logic programs with strong negation, a programmer can state in the language that the negation is classical rather than strong. This can be done simply by adding rules of the form $A \vee \neg A$ for every atom. Moreover, the programmer has the opportunity of stating which negation, strong or classical, is desired for each of the atoms in the language, by choosing whether to add or not, for each atom, such a disjunctive rule.

WFSX and strong negation

Since *WFSX* exhibits all the above mentioned properties of strong negation (cf. section 10.1) and is defined as an extension of WFS, it seems that it should be closely related to stationary semantics with strong negation. In fact:

Theorem 5.1.5 (*WFSX* and strong negation) *If an interpretation M is a stationary*

$$AX_{\neg} = \{\neg\_A \Rightarrow \sim A \mid A \in \mathcal{H}\}$$

model of a program P then M is a WFSX *partial stable model of P.*

Proof: Trivial, given the proof of theorem 5.1.6 below. $\diamondsuit$

Thus *WFSX* gives semantics to more programs and, whenever both semantics give a meaning to a program, the WF model of *WFSX* is a (possibly proper) subset of that of stationary semantics with strong negation. The differences between *WFSX* and stationary semantics with strong negation are best shown with the help of an example.

Example 5.9 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ a \\ b & \leftarrow & a \\ \neg b & & \end{array}$$

which has no strong stationary models. According to *WFSX* its well founded model (and only partial stable model) is:

$$M = \{\neg b, not\ b, not\ \neg a\}.$$

Note that M is not even a model in the (usual) sense of [Przymusinska and Przymusinski, 1990], because for the second rule the truth value of the head (false) is smaller than the truth value of the body (undefined).

Recall that in *WFSX* $\neg$–negation overrides undefinedness (of b in this case). The truth of $\neg L$ is an *explicit* declaration that L is false.

Any semantics complying with proposition 5.1.1 cannot have M as a model of the program: not_b is in an expansion iff $\sim b$ is in all minimal models of that expansion, but if this is the case then (by the second rule clause) $\sim a$ should also be in all minimal models, which would necessarily entail $not\ a$ in the expansion.

In order to reconstruct *WFSX* in the generic schema, a new condition for adding default negation is required, forcing a default literal not_L to assuredly belong to an expansion also in the case where the explicit negation $\neg\_L$ is in all models.

Theorem 5.1.6 (*WFSX* semantics) *An interpretation M is a partial stable model of a program P iff M is a stationary*

$$AX_{\neg} = \{\neg\_A \Rightarrow not\_A \mid A \in \mathcal{H}\}$$

model of P.

Alternatively, M is a partial stable model of P iff M is the meaning of a P^* such that:

$$P^* = \neg\_P \cup \left\{not\_L \mid P^* \models_{CIRC} \sim L \text{ or } P^* \models \neg\_L\right\} \cup \{\sim not\ L \mid P^* \models L\}$$

Proof: In appendix $\diamondsuit$

Example 5.10 The program P of example 5.9 has a single expansion:

$$P^* = \neg\_P \cup \{not\_b, \sim not\_\neg\_b, not\_\neg\_a\}.$$

In fact its minimal models are:

$$\{\ not\_a,\ not\_\neg\_a,\ not\_b,\ \sim not\_\neg\_b,\quad a,\ \sim\neg\_a,\quad b,\ \neg\_b\}$$
$$\{\sim not\_a,\ not\_\neg\_a,\ not\_b,\ \sim not\_\neg\_b,\ \sim a,\ \sim\neg\_a,\ \sim b,\ \neg\_b\}$$

In all these models we have:

- $\sim\neg\_a$ so we must introduce $not\_\neg\_a$;
- $\neg\_b$ so we must introduce $\sim not\_\neg\_b$ and, by the second disjunct, not_b. Note that there is no need for adding not_b in the first alternative of theorem 5.1.6 since it follows as a consequence, given the axioms in $AX_{\neg}$.

The semantics of P is the meaning of P^*, i.e.

$$\{\neg b, not\ b, not\ \neg a\}.$$

giving its *WFSX* single partial model.

5.1.5 Logic programs with ¬-negation and disjunction

Based on the similarities between the generic definition of stationary semantics for extended programs and that of stationary semantics for normal logic programs, it is easy to extend the former for extended disjunctive logic programs based on the extension of the latter for disjunctive normal programs [Przymusinski, 1991b], where the rule syntax is enlarged to include disjunctive conclusions.

First we have to extend the definition of $\neg\_P$ for the case of disjunctive programs. This extension is obtained simply by adjoining to definition 5.1.3:

> "[...] *reinterpreting the connective* $\vee$ *in logic programs as classical disjunction*".

With this new context we define:

Definition 5.1.10 *A stationary* $AX_{\neg}$ *expansion of an extended disjunctive program* P *is any consistent theory* P^* *which satisfies the following fixed point condition (where the distributive axiom* $not\ (A \wedge B) \equiv not\ A \vee not\ B$ *is assumed):*

$$P^* = \neg\_P \cup AX_{\neg} \cup \left\{ not\ F \mid P^* \models_{CIRC} \neg F \right\}$$

where F *is an arbitrary conjunction of positive (resp. negative) objective literals.*

Given this definition the semantics follows similarly to section 5.1.2.

Example 5.11 Consider program P :

$$\begin{array}{rcl} p & \leftarrow & not\ a \\ p & \leftarrow & not\ \neg b \\ a \vee \neg b & & \end{array}$$

and let $AX_{\neg}$ be the axioms for strong negation. The only stationary $AX_{\neg}$ expansion of P is:

$$P^* = \neg\_P \cup AX_{\neg} \cup \{not\_\neg\_a, not\_b, \sim not\_p, not\_\neg\_p, not\_a \vee not\_\neg\_b, \sim not\_a \vee \sim not\_\neg\_b\}.$$

Thus the only stationary $AX_{\neg}$ model is $\{p, not\ \neg p, not\ \neg a, not\ b\}$.

Henceforth, the way is open for the study of the interaction between ¬ and disjunction in semantics of extended programs, and comparisons among those semantics via disjunction. One such result concerning the latter is the comparison between the use of classical or strong negation mentioned above in page 50.

Example 5.12 In example 5.8 it is shown that the program P :

$$\begin{array}{rcl} a & \leftarrow & b \\ \neg a & & \end{array}$$

considering ¬ as classical negation, has the single stationary model:

$$M = \{not\ a, \neg a, not\ b, \neg b\}.$$

This fails to comply with the property of supportedness. There we argue that if one wants the result of M then the program should be written as P_2 :

$$\begin{array}{rcl} a & \leftarrow & b \\ \neg a & & \\ b \vee \neg b & & \end{array}$$

It is easy to see that, with the above definition of stationary expansion of extended disjunctive programs, the only stationary model of P_2, when ¬ is strong negation, is M.

It is known [Przymusinski, 1991b] that a definition such as 5.1.10 makes program disjunctions exclusive. This is seen in example 5.11. In order to treat disjunctions as inclusive rather than exclusive, in non–extended disjunctive programs, it suffices to replace $\models_{CIRC}$ by $\models_{WECWA}$ in the definition of expansions [Przymusinski, 1991b], where $WECWA$ stands for Weak Extended Closed World Assumption [Ross and Topor, 1988] or Weak Generalized Closed World Assumption [Rajasekar *et al.*, 1989].

Further developments on the introduction of disjunction in extended logic programs, including that of inclusive disjunction, are beyond the scope of this work.

5.2 Autoepistemic logics for *WFSX*

In the previous section we identified distinct acceptations of ¬–negation in different semantics for extended logic programs. Some properties of each of those ¬–negations were presented. However no epistemic meaning was given to such ¬–negations.

The main goals of this section are to define an autoepistemic logic capable of expressing extended logic programs, and to study the epistemic meaning of ¬–negation in such programs.

The structure of this section is as follows: we begin by reviewing Moore's autoepistemic logic, introduced in [Moore, 1985], and the autoepistemic logic of closed beliefs introduced in [Przymusinski, 1991a]. In section 5.2.2 we examine the epistemic meaning of a second negation in both logic programs and autoepistemic theories, in terms of logics of knowledge and belief. Then we define autoepistemic logics with explicit negation, and finally we relate them to the *WFSX* semantics of extended logic programs.

Parts of this section appear in [Alferes and Pereira, 1993b].

5.2.1 Moore's and Przymusinski's autoepistemic logics

A *propositional autoepistemic language* is any propositional language *Lang* with the property that for any proposition A in *Lang*, hereafter called *objective*, its alphabet also contains the corresponding *belief proposition* $\mathcal{L}\,A$, i.e. the proposition whose name is a string beginning with the symbol $\mathcal{L}$ followed by A. The intended meaning of $\mathcal{L}\,A$ is *"A is believed"*.

An *autoepistemic theory* is any theory T over an autoepistemic language[5]. The following definition of stable autoepistemic expansion can be easily shown equivalent to Moore's:

Definition 5.2.1 (Stable autoepistemic expansion) *A consistent theory T^* is a stable autoepistemic expansion of the autoepistemic theory T iff:*

- *$T^* = T \cup B$, where B is a (possibly empty) set of belief literals, i.e. literals of the form $\mathcal{L}\,A$ or $\sim\mathcal{L}\,A$, where A is an objective proposition, and*
- *T^* satisfies the following conditions:*

$$\begin{array}{lllll} T^* & \models & \mathcal{L}\,A & \equiv & T^* \models A \\ T^* & \models & \sim\mathcal{L}\,A & \equiv & T^* \not\models A \end{array}$$

This definition expresses positive and negative introspection of a rational agent: an agent believes in some proposition A iff A belongs to all models of its knowledge; and has no reason to believe in A ($\sim\mathcal{L}\,A$) iff A doesn't belong to all models of its knowledge.

Remark 5.2.1 *In the original definition of Moore, the belief operator $\mathcal{L}$ can be applied to any formula, and thus the definition of expansion is modified accordingly. In [Przymusinski, 1991a] it is shown the restriction to propositions in the above definition doesn't influence generality.*

[5] Like in the previous section, we use the symbol $\sim$ to denote classical negation in theories.

Moreover, as our interest is focused on autoepistemic logic for logic programming, such general formulae do not occur in theories (cf. Gelfond's transformation, informally mentioned in the introduction to this chapter and formalized in definition 5.2.6 below).

Example 5.13 Consider the following autoepistemic theory T, modeling the so called birds fly situation:

$$\begin{array}{rcl} bird(X) \wedge \sim\mathcal{L}\, abnormal(X) & \Rightarrow & fly(X) \\ bird(a) & & \\ bird(b) & & \\ ab(b) & & \end{array}$$

Its only stable expansion is (with obvious abbreviations):

$$T \cup \{\mathcal{L}\, b(a), \mathcal{L}\, b(b), \mathcal{L}\, ab(b), \mathcal{L}\, f(a), \sim\mathcal{L}\, ab(a), \sim\mathcal{L}\, f(b)\}$$

stating that an agent with knowledge T believes that a and b are birds, b is an abnormal bird and a flies, and has no reason to believe that a is abnormal, and that b flies.

Of course, some autoepistemic theories might have several stable expansions:

Example 5.14 The theory T :

$$\begin{array}{rcl} a & \vee & \mathcal{L}\, b \\ b & \vee & \mathcal{L}\, a \end{array}$$

has two expansions, namely:

$$\begin{array}{c} T \cup \{\mathcal{L}\, a, \sim\mathcal{L}\, b\} \\ T \cup \{\mathcal{L}\, b, \sim\mathcal{L}\, a\} \end{array}$$

Each of these can be envisaged as a belief state, i.e. an agent with knowledge T has two possible states of belief: either he believes in a and in that case has no reason to believe in b, or vice–versa. A sceptical agent with these belief states should have no reason to believe nor disbelieve neither a nor b.

In [Przymusinski, 1991a] Przymusinski argues, and we concur, that Moore's autoepistemic logic has some important drawbacks:

• First, quite reasonable theories have no stable expansions [Morris, 1988, Przymusinski, 1989c]. For example the theory:

$$\begin{array}{c} broken\_car \\ can\_fix\_it \vee \mathcal{L}\, can\_fix\_it \end{array}$$

has no stable expansion, because no consistent addition of beliefs to the theory entail believing can_fix_it, and disbelieving that it can be fixed leads to an inconsistency[6], the agent should rest agnostic about that, neither believing nor disbelieving it can be fixed. However, one expects a reasoner with this knowledge at least to believe that the car is broken.

• Another important drawback is that, even for theories with stable expansions, Moore's autoepistemic logic does not always lead to the expected intended semantics. For instance consider the example[7]:

[6]Note that by adding $\sim\mathcal{L}\, can\_fix\_it$ to the theory, can_fix_it follows as a consequence, and thus $\mathcal{L}\, can\_fix\_it$ must be added (inconsistency).

[7]This example first appeared in [Bonatti, 1993], in the form of a logic program.

Example 5.15 A robot is programmed to carry some money from bank 1 to bank 2. There are two possible routes, denoted a and b; the robot chooses one of them, provided that it has no reason to believe there is trouble along the route. If it can choose any route then it should prefer route a. After choosing a route, the robot signals *" I'm leaving"* and tries to reach bank 2. This task can be naturally formalized by the autoepistemic theory:

$$\begin{array}{rcl} \mathcal{L}\, trouble(a) \wedge \sim\mathcal{L}\, trouble(b) & \Rightarrow & choose(b) \\ \mathcal{L}\, trouble(b) \wedge \sim\mathcal{L}\, trouble(a) & \Rightarrow & choose(a) \\ \sim\mathcal{L}\, trouble(a) \wedge \sim\mathcal{L}\, trouble(b) & \Rightarrow & choose(a) \\ choose(a) & \Rightarrow & signal \\ choose(b) & \Rightarrow & signal \end{array}$$

Given this knowledge, its unique stable expansion captures the intended meaning, i.e. the robot has no reason to believe that there is trouble in any of the routes, and thus chooses route a and signals.

Supposed now one adds to the theory the knowledge that there is some trouble in one of the routes, but it is not known which, expressed by:

$$trouble(a) \vee trouble(b)$$

The resulting theory has two stable expansions, both of which contain $\mathcal{L}\, signal$, and where one contains $\mathcal{L} choose(a)$ and the other contains $\mathcal{L} choose(b)$. According to the stable expansions a sceptical reasoner would believe neither in $choose(a)$ nor in $choose(b)$, i.e. the robot wouldn't choose any of the routes, which is reasonable. However such a reasoner would believe in $signal$, i.e. the robot says *" I'm leaving"*, which clearly doesn't express the intended meaning.

• Stable expansions cannot be effectively computed even within simple classes of theories, such as propositional logic programs [Kautz and Selman, 1989]. This is an important drawback, specially if one is interest in a theory for implementing knowledge representation and reasoning.

• Last but not least, by always insisting on completely deciding all of an agent's beliefs, stable expansions often lack expressibility. This issue will be further explored in this section.

In order to overcome these drawbacks Przymusinski introduced in [Przymusinski, 1991a] the general notion of autoepistemic logics of closed beliefs, and presented the circumscriptive autoepistemic logics as an important special case.

The notion of autoepistemic logics of closed beliefs arises naturally as a generalization of Moore's autoepistemic logics. First Przymusinski points out that in the definition of stable expansion, $T^* \not\models A$ can be replaced by $T^* \models_{CWA} \sim A$, and proceeds to argue that stable expansions are a special case of expansions based on the general notions of positive and negative introspection.

Definition 5.2.2 (Autoepistemic expansion) *A consistent theory T^* is an autoepistemic expansion of a theory T iff*

- *$T^* = T \cup B$, where B is a (possibly empty) set of belief literals, i.e. literals of the form $\mathcal{L}\, A$ or $\sim\mathcal{L}\, A$, where A is an objective proposition, and*
- *T^* satisfies the following conditions:*

$$\begin{array}{rcl} T^* \models \mathcal{L}\, A & \equiv & T^* \models_{op} A \\ T^* \models \sim\mathcal{L}\, A & \equiv & T^* \models_{cl} \sim A \end{array}$$

where $\models_{op}$ is a general entailment operator of open beliefs (or positive introspection) and $\models_{cl}$ is a general entailment operator of closed beliefs (or negative introspection).

Depending on the chosen positive and negative introspection entailment operators different autoepistemic logics are obtained.

Based on this general definition, Przymusinski defines Circumscriptive Expansions simply by choosing $\models$ as the positive and $\models_{CIRC}$ [8] as the negative introspection operators. He also shows that with this definition of expansion, all of the above pointed out drawbacks are overcome, and that, through Gelfond's transformation between normal logic programs and autoepistemic theories (whereby *not* L is construed as $\sim\mathcal{L}\,L$), the least expansion is equivalent to the well–founded semantics of [Gelder *et al.*, 1991].

5.2.2 Why ¬ should not be classical negation

In [Lifschitz, 1991c, Lifschitz, 1991b] Lifschitz describes the nonmonotonic logic MBNF, based on the notion of minimal knowledge[9] and the ideas of justification and negation as failure (or default negation).

Formulae of the propositional MBNF are built from propositional symbols and standard connectives, plus two modal operators: $\mathcal{B}$ and *not* . Intuitively, for any proposition l, $\mathcal{B}\,l$ expresses that l is minimally known, and *not* l that l cannot be assumed or, alternatively, that there is no reason to believe in l.

This language has great expressivity, allowing for formulae combining truth in the *world* with truth in the knowledge of an agent. For example:

$$a \vee \mathcal{B}\,a$$

states that a is true in the real world or a is known as true by the agent.

For this language Lifschitz defines a model theory and a consequence relation. Moreover he relates a special class of this logic with logic programming, and argues that each normal logic program rule:

$$H \leftarrow A_1, \ldots, A_n, not\ B_1, \ldots not\ B_m$$

where H, A_i and B_j are atoms, should be translated into (where $\Rightarrow$ is material implication):

$$\mathcal{B}\,A_1, \ldots, \mathcal{B}\,A_n, not\ B_1, \ldots not\ B_m \Rightarrow \mathcal{B}\,H$$

In this translation he is assuming, and we concur, that a logic program only speaks of propositions known by the agent and his beliefs or disbeliefs (it will be seen that *not* $\equiv \sim\mathcal{L}$), and not of truth in the world. Thus the above rule is reads:

"If A_1 is known to be true by the agent, and ..., and A_n is known to be true by the agent, and the agent has no reason to believe in B_1, and ..., and the agent has no reason to believe in B_m, then H is known to be true by the agent."

Because $\mathcal{B}$ affects all atoms without exception, it is simply dropped in logic programs.

We argue that this reading is also applicable to extended logic programs. For instance the program:

$$\begin{array}{lcl} p & \leftarrow & a \\ p & \leftarrow & \neg a \end{array}$$

should be read in MBNF as:

[8] Here $\models_{CIRC}$ is as in the previous section (cf. page 43), but where the fixed propositions are of the form $\mathcal{L}\,A$ instead of *not_A*.

[9] Minimal knowledge (or maximal ignorance) was formalized earlier by several authors, including [Konolige, 1982, Shoham, 1986, Lin, 1988, Lin and Shoham, 1990].

"If a is known to be true by the agent, then p is known to be true by the agent. If ∼a is known to be true by the agent (i.e. a is known to be false) then p is known to be true by the agent." and represent in MBNF by:

$$\begin{array}{rcl} \mathcal{B}\,a & \Rightarrow & \mathcal{B}\,p \\ \mathcal{B}\,{\sim}a & \Rightarrow & \mathcal{B}\,p \end{array}$$

Clearly $\mathcal{B}\,a \vee \mathcal{B}\,{\sim}a$ is not a tautology, since it is not always the case that the agent knows that a is true or knows that a is false[10]. Situations occur where the agent, based on his knowledge, cannot conclude neither $\mathcal{B}\,a$ nor $\mathcal{B}\,{\sim}a$. Thus $\mathcal{B}\,p$ should not be a consequence of the program.

Note how, with this epistemic reading, the clausal contrapositives of logic program's rules are not implicit. For example the rule $p \leftarrow \neg a$ does not implicitly add its contrapositive $a \leftarrow \neg p$. This happens because the first rule is translated into $\mathcal{B}\,{\sim}a \Rightarrow \mathcal{B}\,p$, stating that *"p is known to be true if a is known to be false"*. The latter's contrapositive, implicit in the use of material implication, is ${\sim}\mathcal{B}\,p \Rightarrow {\sim}\mathcal{B}\,{\sim}a$, and states that *" a is not known to be false if p is not known to be true"*, which is clearly not equivalent to the logic program's rule contrapositive (nor to any other possible rule). The program rule's contrapositive translates to $\mathcal{B}\,{\sim}p \Rightarrow \mathcal{B}\,a$, stating that *"a is known to be true if p is known to be false"*.

The issue of not using rule contrapositives in logic programs is further explored in chapter 6 where, by relating logic programs with default theories, rules of the former are viewed as inference rules like in the latter.

An alternative translation to MBNF, but deriving $\mathcal{B}\,p$, would be:

$$\begin{array}{rcl} \mathcal{B}\,a & \Rightarrow & \mathcal{B}\,p \\ {\sim}\mathcal{B}\,a & \Rightarrow & \mathcal{B}\,p \end{array}$$

stating that:

"If a is known to be true by the agent, then p is known to be true by the agent. If a is not known to be true by the agent then p is known to be true by the agent."

With this representation, as $\mathcal{B}\,a \vee {\sim}\mathcal{B}\,a$ is clearly a tautology, $\mathcal{B}\,p$ is a consequence of the program. However note how this representation fails to concur with the principle that a logic program only speaks of propositions known true or false by an agent, by speaking overtly of what an agent doesn't know.

Moreover the first translation is more general because it can achieve the effect of the second by stating that for proposition a the agent is omniscient (i.e. knows its truth or falsity), by adding to the theory a clause of the form:

$$\mathcal{B}\,a \vee \mathcal{B}\,{\sim}a$$

If one agrees with the first translation, all propositions are implicitly preceded by the knowledge operator $\mathcal{B}$. For simplicity it has been omitted. Nevertheless one must take care with the translation of $\mathcal{B}\,{\sim}A$ into logic programs. If one translates it into ${\sim}A$ then one runs into the problem of the second translation because $\mathcal{B}\,A \vee \mathcal{B}\,{\sim}A$ is translated into the tautology $A \vee {\sim}A$. In order not to turn it into a tautology we're justified to introduce a new negation symbol $\neg$ (*explicit negation*).

The simplification into logic programs is then obtained by replacing each proposition of the form:

- $\mathcal{B}\,A$, where A is an atom, simply by A;
- $\mathcal{B}\,{\sim}A$ by $\neg A$.

[10] Note the difference from $\mathcal{B}\,(a \vee {\sim}a)$, stating that the agent knows that a is true or false.

By $\neg A$ we designate the explicit negation of A, standing for knowing that A is false, i.e. $\neg \equiv \mathcal{B} \sim$.

The arguments above also apply to the translation from autoepistemic logic to logic programs, and from autoepistemic logics with $\mathcal{B}$ to autoepistemic logics without $\mathcal{B}$.

This leads to an extension of the definition of autoepistemic theory of section 5.2.1 above:

Definition 5.2.3 (Autoepistemic theory) *A* propositional autoepistemic language *is any propositional language Lang with the property that for any objective proposition A in Lang its alphabet also contains the corresponding proposition $\neg A$, and* belief propositions $\mathcal{L}\, A$ *and* $\mathcal{L}\, \neg A$. *Propositions $\neg A$ (resp. $\mathcal{L}\, A$ and $\mathcal{L}\, \neg A$) are to be understood as ones whose name is a string beginning with the symbol $\neg$ (resp. $\mathcal{L}$ and $\mathcal{L}\, \neg$) followed by A. The intended meaning of $\neg A$ is that* "A is known to be false"*, and that of $\mathcal{L}\, A$ is* "A is believed"*.*

An autoepistemic theory *is any theory T over an autoepistemic language.*

The definition and study of expansions in these theories is made in the next section.

With these ideas in mind the epistemic coherence principle follows intuitively: It is clear that $\mathcal{B} \sim A$ (i.e. $\neg A$) should entail $\sim \mathcal{L}\, A$, i.e. if a rational agent knows that A is false then this knowledge is enough to conclude that he does not believe A is true.

This principle provides for the possibility of expressing in a theory that some proposition A should not be believed by the agent, simply by introducing the factual knowledge $\neg A$. Note that with the autoepistemic theories described above, and with the restriction that a theory should speak only of propositions known by the agent, this is impossible. One can state that some proposition A is believed by adding to the theory the factual knowledge A. Indeed since

$$T^* \models \mathcal{L}\, A \equiv T^* \models A$$

and T^* is obtained from T only by adding belief propositions, it follows that if $A \in T$ then A is in all models of T^* and $\mathcal{L}\, A$ is clearly true.

The addition of $\sim A$ does not comply with the said principle. We've seen it is a simplification of $\sim \mathcal{B}\, A$, and states that A is not known true by the agent, not that he doesn't believe A. Facts of the form $\sim \mathcal{L}\, A$ do not provide the intended result because for a theory to be an expansion the equivalence

$$T^* \models \sim \mathcal{L}\, A \equiv T^* \models_{CIRC} \sim A$$

must hold, and $\sim \mathcal{L}\, A$ does not provide a way of imposing $\sim A$ on all minimal models.

A fact $\neg A$, stating that A is known to be false, accords with the principle that a theory should only speak of propositions known by the agent and, via coherence, provides a natural instrument for imposing the "disbelief" of a proposition.

5.2.3 Autoepistemic logic with explicit negation

Here we define expansions and the semantics of autoepistemic theories with explicit negation (i.e. theories as per definition 5.2.3 above).

The language of these theories has the advantage, over the previous autoepistemic languages, that it is possible to express in them knowledge about falsity, or negative information. In the others only absence of positive information about the knowledge of an agent is expressible. We mark the difference between these two notions with the help of an example:

Stating that:

(1) *I know that Susan is not married to Peter*

is clearly not the same as stating that:

(2) *I do not know that Susan is married to Peter.*

While autoepistemic theories with explicit negation can express (1) and (2)[11], other autoepistemic languages can express (2) but cannot express (1).

We argue that not being able to express (1) is a serious limitation. Not being able to express (2), as in logic programming, results naturally if one agrees that a program speaks only of what it knows; never about what it doesn't know. Presently, in logic programs only positive knowledge is grounded. The ability to express both (1) and (2) also allows the absence of positive or negative information to be conveyed: $\sim\mathcal{B}\, p \wedge \sim\mathcal{B}\ \sim p$, i.e. in the language of autoepistemic theories with explicit negation:

$$\sim p \wedge \sim\neg p$$

The notion of expansion is a generalization, for the new language of theories, of circumscriptive autoepistemic logic.

Recall that an autoepistemic theory represents the knowledge of a reasoning agent where:

- A stands for: the truth of A is known by the agent;
- $\neg A$ stands for: the falsity of A is known by the agent;
- $\mathcal{L}\ A$ stands for: A is believed by the agent;
- $\mathcal{L}\,\neg A$ stands for: $\neg A$ is believed by the agent;
- $\sim\mathcal{L}\ A$ stands for: A is not believed by the agent;
- $\sim\mathcal{L}\,\neg A$ stands for: $\neg A$ is not believed by the agent.

Under CWA, A is not believed if there is no reason to believe A. But A may be not believed as an hypothesis which may or may not be held, even if there is no reason to believe A.

In order to define expansion in accordance with the autoepistemic logic of closed beliefs we have to define the positive and negative entailment operators, i.e. what are the introspection mechanisms that allow the agent to derive his beliefs, and to derive his disbeliefs.

For positive introspection, we say an agent believes in a proposition A if and only if A belongs to all models of the theory. In other words A is believed by the agent iff A is known to be true by him.

For negative introspection, we say an agent disbelieves a proposition A iff $\sim A$ belongs to all minimal models of the theory, or $\neg A$ belongs to all models of the theory. In other words, A is disbelieved iff $\sim A$ is minimally entailed by the theory or is known to be false.

Accordingly:

Definition 5.2.4 (Autoepistemic expansions) *A consistent theory T^* is an autoepistemic expansion of a theory T iff*

- *$T^* = T \cup B$, where B is a (possibly empty) set of belief literals, i.e. literals of the form $\mathcal{L}\,A$, $\mathcal{L}\,\neg A$, $\sim\mathcal{L}\,A$, or $\sim\mathcal{L}\,\neg A$, where A is an objective proposition, and*
- *T^* satisfies the following conditions:*

$$\begin{array}{llll}
T^* \models \mathcal{L}\ A & \equiv & T^* \models A & \\
T^* \models \mathcal{L}\,\neg A & \equiv & T^* \models \neg A & \\
T^* \models \sim\mathcal{L}\ A & \equiv & T^* \models_{CIRC} \sim A & \text{or } T^* \models \neg A \\
T^* \models \sim\mathcal{L}\,\neg A & \equiv & T^* \models_{CIRC} \sim\neg A & \text{or } T^* \models A
\end{array}$$

where $\models_{CIRC}$ is as defined in 5.1.5 but where fixed propositions are of the form $\mathcal{L}\,L$ instead.

[11] Note that in our definition of theories it is still possible to have classically negated literals $\sim L$ which stand, as seen above, for $\sim\mathcal{B}\,L$.

Example 5.16 Consider the following autoepistemic theory T, which is a modification of the birds fly situation of example 5.13:

$$\begin{array}{rcl} \sim\mathcal{L}\,\neg fly(X) \wedge bird(X) & \Rightarrow & fly(X) \\ bird(a) & & \\ bird(b) & & \\ \neg fly(b) & & \end{array}$$

where the last clause expresses that b is known not to fly.

Its only expansion is (with obvious abbreviations):

$$T \cup \{\mathcal{L}\,b(a), \mathcal{L}\,b(b), \mathcal{L}\,f(a), \mathcal{L}\,\neg f(b), \sim\mathcal{L}\,\neg b(a), \sim\mathcal{L}\,\neg b(b), \sim\mathcal{L}\,\neg f(a), \sim\mathcal{L}\,f(b)\}$$

stating that an agent with knowledge T believes that a and b are birds, b doesn't fly, a flies, and disbelieves that a and b are not birds, that a doesn't fly, and that b flies.

Example 5.17 [12] Consider an agent with the following knowledge:

- *Peter is a bachelor;*
- *A man is known not to be married if he is known to be a bachelor;*
- *Susan is known to be married to Peter, if we do not believe she's married to Tom.*
- *Susan is known to be married to Tom, if we do not believe she's married to Peter.*
- *It is known that no one is married to oneself.*

rendered by the autoepistemic theory T (with obvious abbreviations):

$$\begin{array}{rcl} b(p) & & \\ b(X) & \Rightarrow & \neg m(X,Y) \\ \sim\mathcal{L}\,m(t,s) & \Rightarrow & m(p,s) \\ \sim\mathcal{L}\,m(p,s) & \Rightarrow & m(t,s) \\ \neg m(X,X) & & \end{array}$$

The only expansion of T contains, among others, the belief propositions:

$$\{\mathcal{L}\,b(p), \mathcal{L}\,\neg m(p,s), \sim\mathcal{L}\,m(p,s), \mathcal{L}\,m(t,s)\}$$

Note how explicit negation imposes, via the new disjunct of negative introspection, that the knowledge that Peter is not married to Susan imposes the disbelieve he is married to her, thus allowing for the conclusion that Tom is married to Susan.

In both the above examples all of an agent's beliefs are completely decided, in the sense that for any proposition A the agent either believes or disbelieves A. Of course, since these expansions result from a generalization of circumscriptive expansions, this is not in general the case.

Example 5.18 Consider the statements:

- *It is known or believed that one can fix the car.*
- *If it is not believed that one can fix the car then it is known that an expert is called for.*
- *It is known that an expert is not called for.*

[12] This example first appeared in [Wagner, 1993], in the form of a logic program.

rendered by the autoepistemic theory T :

$$\begin{array}{rcl} can\_fix\_car \vee \mathcal{L}\, can\_fix\_car & & \\ \sim\mathcal{L}\, can\_fix\_car & \Rightarrow & call\_expert \\ \neg call\_expert & & \end{array}$$

The only expansion of T is:

$$T \cup \{\mathcal{L}\,\neg call\_expert, \sim\mathcal{L}\, call\_expert\}$$

stating that an agent believes that an expert is not called and that he disbelieves an expert is called.

Note that about can_fix_car the agent remains undefined, neither believing nor disbelieving it. This is due to, on the one hand, that believing it is impossible since it is not a consequence of all models, and so it cannot be derived by positive introspection; on the other hand disbelieving it leads to an inconsistency.

The result of this expansion correctly describes the intuitive meaning of the above piece of knowledge: can_fix_car is not believed nor disbelieved. Since the agent knows that an expert is not called for this certainly plays a more important rôle than the undefinedness of can_fix_car, and the latter does not interfere with not believing that an expert is called. The knowledge that an expert is not called for is instrumental for disbelieving that it is called for. Mark here the similarities with the new condition on the definition of models in logic programs, where explicit negation overrides any undefined conclusion following from undefinedness in the body of rules.

By always insisting on completely deciding all of an agent's beliefs stable expansions do not give a meaning to T. Circumscriptive autoepistemic logics too cannot express this knowledge and give the expected result. In fact, even if one replaces $\neg$ by $\sim$in the last clause, and applies circumscriptive autoepistemic logic, no circumscriptive expansion exists. Technically this happens because, by the last clause, $\sim call\_expert$ belongs to all models; thus, by the second clause, $\mathcal{L}\, can\_fix\_car$ also belongs to all models; in any expansion this belief proposition can be true only if can_fix_car belongs to all models, which is not the case.

Intuitively, this happens because on the one hand $\sim call\_expert$ is interpreted as "the agent doesn't know that an expert is called for", and thus he must believe that he can fix the car. One the other hand he cannot believe he can fix the car because that is not a result of positive introspection.

Like Moore's autoepistemic theories, autoepistemic theories with explicit negation might have several expansions:

Example 5.19 Consider the theory T, describing the so–called Nixon diamond situation:

$$\begin{array}{rcl} republican(nixon) & & \\ quaker(nixon) & & \\ republican(X), \sim\mathcal{L}\, pacifist(X) & \Rightarrow & \neg pacifist(X) \\ quaker(X), \sim\mathcal{L}\,\neg pacifist(X) & \Rightarrow & pacifist(X) \end{array}$$

T has three expansions, namely:

$$\begin{array}{lll} T & \cup & \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), \mathcal{L}\, p(n), \ \sim\mathcal{L}\,\neg r(n), \sim\mathcal{L}\,\neg q(n), \sim\mathcal{L}\,\neg p(n)\} \\ T & \cup & \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), \mathcal{L}\,\neg p(n), \sim\mathcal{L}\,\neg r(n), \sim\mathcal{L}\,\neg q(n), \sim\mathcal{L}\, p(n)\} \\ T & \cup & \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), \qquad\quad \sim\mathcal{L}\,\neg r(n), \sim\mathcal{L}\,\neg q(n)\} \end{array}$$

The first states that it is believed that Nixon is a pacisfist; the second that it is believed that Nixon is not a pacifist; and the third remains undefined in what concerns Nixon being or not a pacisfist.

When confronted with several expansions (i.e. several possible states of beliefs) a sceptical reasoner should only conclude what is common to all. Here that coincides with the third expansion.

In point of fact, the intersection of all expansions is itself an expansion:

Theorem 5.2.1 (Least expansion) *The intersection of all expansions of an autoepistemic theory with explicit negation is itself an expansion.*

Proof: Follows directly from the relation with *WFSX* and the properties of the latter. $\diamondsuit$

Up to now we've defined expansions but did not use them to formally assign a meaning to a theory[13]. Now we define the meaning of an autoepistemic theory with explicit negation in a quite straightforward way.

The meaning of an expansion is the set of belief (and disbelief) propositions in it. In other words, each expansion corresponds to a belief state, and its meaning is the set of beliefs and disbeliefs of the agent in that state. The meaning of an autoepistemic theory is the set of all meanings of all expansions.

As noted above, when several expansions exist a sceptical reasoner should only conclude what is common to all. In order to easily model such a reasoner we also define a sceptical meaning of theories.

Definition 5.2.5 (Meaning of an autoepistemic theory) *Let T be an autoepistemic theory with expansions $T \cup B_k$[14] such that $k \in K$.*

The meaning of T is determined by:

$$\{B_k \mid k \in K\}$$

The sceptical meaning of T is determined by:

$$\bigcap_{k \in K} B_k$$

Directly from theorem 5.2.1 it follows that:

Theorem 5.2.2 (Sceptical meaning of a theory) *Let T be an autoepistemic theory with explicit negation, whose least expansion is:*

$$T \cup B$$

The sceptical meaning of T can be equivalently defined as B.

Example 5.20 The meaning of the Nixon Diamond program of example 5.19 is:

$$\left\{ \begin{array}{l} \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), \mathcal{L}\, p(n), {\sim}\mathcal{L}\, \neg r(n), {\sim}\mathcal{L}\, \neg q(n), {\sim}\mathcal{L}\, \neg p(n)\} \\ \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), \mathcal{L}\, \neg p(n), {\sim}\mathcal{L}\, \neg r(n), {\sim}\mathcal{L}\, \neg q(n), {\sim}\mathcal{L}\, p(n)\} \\ \qquad \{\mathcal{L}\, r(n), \mathcal{L}\, q(n), {\sim}\mathcal{L}\, \neg r(n), {\sim}\mathcal{L}\, \neg q(n)\} \end{array} \right\}$$

and its sceptical meaning is:

$$\{\mathcal{L}\, r(n), \mathcal{L}\, q(n), {\sim}\mathcal{L}\, \neg r(n), {\sim}\mathcal{L}\, \neg q(n)\}$$

[13] However, in the examples, we informally presented the meaning of theories based on their expansions.

[14] Note that each B_k is a set of belief propositions.

5.2.4 Autoepistemic belief revision

Not all theories have a meaning:

Example 5.21 The theory T :

$$\begin{array}{r} a \\ \neg a \end{array}$$

has no meaning. This is reasonable since T is clearly contradictory.

Example 5.22 Consider yet another modification of the birds fly situation of example 5.13:

$$\begin{array}{rcl} \sim_{\mathcal{L}} abnormal(X), bird(X) & \Rightarrow & fly(X) \\ bird(a) & & \\ bird(b) & & \\ \neg fly(b) & & \end{array}$$

This autoepistemic theory has no expansion, and thus no meaning because, on the one hand, knowing that b doesn't fly entails, by positive introspection, that it is believed not to fly and, by negative introspection, that it is disbelieved it does fly; on the other hand, as there are no clauses for $abnormal(b)$, by negative introspection one concludes there is no support for the belief that b is abnormal and thus, since it is known to be a bird it is believed it flies.

However, it can be argued that the contradiction between believing b flies and having no reason to believe it does should not appear at all since it is based on the result of a negative introspection, namely of $\sim_{\mathcal{L}} abnormal(b)$. If a more sceptical definition of the negative introspection operator were used instead, not allowing for $\sim_{\mathcal{L}} abnormal(b)$ to be concluded, then the contradiction would no longer appear. The rationale for not concluding $\sim_{\mathcal{L}} abnormal(b)$ would be:

" if the result of an introspection leads to an inconsistency then revise your introspection".

One possibility to define such a sceptical semantics would be to weaken the conditions of expansion in what regards negative introspection. As a first approximation one could replace the last two conditions of definition 5.2.4 by:

$$\begin{array}{lllllllllll} T^* & \models & \sim_{\mathcal{L}} A & \Rightarrow & T^* & \models_{CIRC} & \sim A & \text{or} & T^* & \models & \neg A \\ T^* & \models & \sim_{\mathcal{L}} \neg A & \Rightarrow & T^* & \models_{CIRC} & \sim \neg A & \text{or} & T^* & \models & A \end{array}$$

i.e. an agent can only disbelieve some proposition A if A is minimally entailed by the theory or A is known to be false. Nevertheless, even if these conditions hold it is not mandatory for the agent to disbelieve A.

Of course this definition is too weak: with it all programs have one expansion without any disbelieved propositions. The idea is to allow $\sim_{\mathcal{L}} A$ propositions not to be derived just in case they are a cause of contradiction. A condition must be introduced in order to obtain the "only if" part of the above implications:

$$\begin{array}{lllllllllllll} T^* & \models & \sim_{\mathcal{L}} A & \Leftrightarrow & (T^* & \models_{CIRC} & \sim A & \text{or} & T^* & \models & \neg A) & \text{and} & Cond \\ T^* & \models & \sim_{\mathcal{L}} \neg A & \Leftrightarrow & (T^* & \models_{CIRC} & \sim \neg A & \text{or} & T^* & \models & A) & \text{and} & Cond \end{array}$$

By the equivalence between this autoepistemic logic and *WFSX*, presented below, the study of such conditions is tantamount to the issue of contradiction avoidance in logic programs with *WFSX*. This issue is studied in length in chapter 8 for logic programs, and is thus not further explored in this section for the corresponding autoepistemic theories.

Another possibility for the definition of such a more sceptical semantics is to add an extra mechanism of introspection on top of the existing ones. This mechanism of introspection would *"remove"* some of the disbeliefs (derived from the existing introspection mechanisms) in order to guarantee noncontradiction.

Again, by the equivalence between this autoepistemic logic and *WFSX*, the study of extra introspection mechanisms is tantamount to contradiction removal in logic programming as studied in chapter 8.

5.2.5 Relation to *WFSX*

In this section we relate autoepistemic theories with explicit negation to *WFSX*.

The relationship between autoepistemic logics and extended logic programs established here brings new mutual benefits to both. For one thing, autoepistemic logics clarify the semantics of extended logic programs providing for a better understanding of the ¬–negation. It also clarifies the use of extended logic programs for representing knowledge and belief.

For another, autoepistemic logics benefit from the existing procedures of extended logic programs. Moreover, the relations between autoepistemic logics and other nonmonotonic formalisms (e.g. defaults, abduction, belief revision) can now be made via logic programs for a broader class because, as we show in the next chapters, extended logic programs relate to all such formalisms.

We now proceed to formalize the relationship between the autoepistemic logic defined above (definition 5.2.4) and *WFSX*.

Definition 5.2.6 (Autoepistemic theories and logic programs) *The autoepistemic theory T corresponding to the extended logic program P is obtained from P by replacing every default literal not L in P by $\sim\mathcal{L}\, L$, and then replacing the rule connective of logic programs by material implication.*

Example 5.23 The theory T corresponding to the logic program P:

$$\begin{array}{rcl} a & \leftarrow & not\ a, \neg b \\ \neg b & \leftarrow & not\ \neg a, c \\ c & & \end{array}$$

is:

$$\begin{array}{rcl} \sim\mathcal{L}\, a, \neg b & \Rightarrow & a \\ \sim\mathcal{L}\, \neg a, c & \Rightarrow & \neg b \\ c & & \end{array}$$

From the definition of autoepistemic expansions (definition 5.2.4) and theorem 5.1.6 it follows easily that:

Theorem 5.2.3 *Let T be the autoepistemic theory corresponding to the extended logic program P. Then*

$$T^* = T \cup \{\mathcal{L}\, A_1, \ldots \mathcal{L}\, A_n, \sim\mathcal{L}\, B_1, \ldots, \sim\mathcal{L}\, B_m\}$$

is an expansion of T, where all the A_i and B_j are objective propositions, iff

$$M = \{A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m\}$$

is a partial stable model of P according to WFSX.

Example 5.24 The only expansions of T of example 5.23 is:

$$T \cup \{\mathcal{L}\, c, \mathcal{L}\, \neg b, \sim\mathcal{L}\, \neg a, \sim\mathcal{L}\, b, \sim\mathcal{L}\, \neg c\}$$

stating that c is believed to be true, b is believed to be false, and a is disbelieved.

The only partial stable model of P is:

$$\{c, \neg b, not\ \neg a, not\ b, not\ \neg c\}$$

Note how the meaning of T relates straightforwardly to the model of P.

Another immediate result (given theorem 5.1.4) is that Przymusinski's stationary expansions of extended logic programs [Przymusinski, 1991b] correspond to the autoepistemic expansions of the theory obtained by first replacing in the program every objective literal $\neg L$ by $\sim L$ and then applying the transformation of definition 5.2.6. Thus, as argued in section 5.2.2, in *WFSX* $\neg A$ should be read "A is known to be false", whereas in stationary expansions it should be read "A is not known to be true".

As suggested by theorem 5.1.2, answer–sets semantics [Gelfond and Lifschitz, 1990] can be obtained by replacing, in definition 5.2.4, $CIRC$ by CWA.

By virtue of theorem 5.2.3, all the arguments found in favour of the definition of autoepistemic expansions apply equally vis–à–vis *WFSX* and the other logic programming approaches that include a second kind of negation.

Chapter 6

WFSX as default logic

A relationship between logic programs and default theories was first proposed in [Bidoit and Froidevaux, 1987] and [Bidoit and Froidevaux, 1988]. The idea is to translate every program rule, into a default one and then compare the extensions of the default theory with the semantics of the corresponding program.

The main motivations for such a relationship are, on the one hand, the use of logic programming as a framework for nonmonotonic reasoning and, on the other hand, the computation of default logic extensions by means of logic programming implementations algorithms. Moreover, by having the relationship established for some semantics of logic programs, it is important to keep with such a relationship with mutual clarifications.

In [Bidoit and Froidevaux, 1988] stable model semantics [Gelfond and Lifschitz, 1988] was shown equivalent to a special case of default theories in the sense of Reiter [Reiter, 1980]. This result was generalized in [Gelfond and Lifschitz, 1990] to programs with explicit negation and answer-set semantics, where they make explicit negation correspond to classical negation used in default theories.

Well Founded Semantics for Default Theories [Baral and Subrahmanian, 1990] extends Reiter's semantics of default theories, resolving some issues of the latter, namely that some theories have no extension and also that some theories have no least extension. Based on the way such issues were resolved in [Baral and Subrahmanian, 1991], the well founded semantics for programs without explicit negation was shown by them equivalent to a special case of the extension classes of default theories in the sense of [Baral and Subrahmanian, 1990]. It turns out that in attempting to directly extend this result to extended logic programs with explicit negation one gets some unintuitive results and no semantics of such logic programs relates to known default theories.

To overcome that, here we first identify principles a default theory semantics should enjoy to that effect, and introduce a default theory semantics that extends that of [Baral and Subrahmanian, 1991] to the larger class of logic programs, but complying with those principles.

Such a relationship to a larger program class improves the cross–fertilization between logic programs and default theories, since we generalize previous results concerning their relationship [Baral and Subrahmanian, 1990, Baral and Subrahmanian, 1991, Bidoit and Froidevaux, 1987, Bidoit and Froidevaux, 1988, Gelfond and Lifschitz, 1990, Przymusinska and Przymusinski, 1991, Przymusinska and Przymusinski, 1993]. Moreover, there is an increasing use of logic programming with explicit negation as a nonmonotonic reasoning tool [Gelfond and Lifschitz, 1990, Pereira *et al.*, 1991d, Pereira *et al.*, 1991f, Pereira *et al.*, 1991g, Pereira *et al.*, 1993d, Pereira *et al.*, 1993e, Wagner, 1991a], which can thus be a vehicle for implementing default theories as well. The relationship also further clarifies the meaning of logic programs combining both explicit negation and negation by default. In particular, it shows in what way explicit

negation corresponds to classical negation in our default theory, and elucidates the use of rules in extended logic programs. Like defaults, rules are unidirectional, so their contrapositives are not implicit: the rule connective $\leftarrow$ is not material implication but has rather the flavour of an inference rule.

Implementationwise, since *WFSX* is definable by a monotonic fixpoint operator, it has desirable computational properties, including top–down and bottom–up procedures. As the default semantics is sound with respect to Reiter's default semantics, whenever an extension exists, we thus provide sound methods for computing the intersection of all extensions for an important subset of Reiter's default theories.

The semantics for default theories presented here is restricted to the language where prerequisites and justifications are finite sets of ground literals, the conclusion is a literal, and all formulas not in default rules are literals as well. Note that when relating defaults to logic programming in the usual way, the language of theories corresponding to programs is already thus restricted. Furthermore, in section 6.6 we show that default theories with this language restriction are nevertheless as powerful as logic programs with explicit negation. Nevertheless, this semantics can be extended to the class of all default theories using methods similar to those presented here. Indeed, D. Pearce extends the default semantics presented here to the class of all default theories, using constructive logic [Pearce, 1992a].

In this chapter we present a semantics for default theories, and show its relationship with *WFSX*. Based on this relationship, we give an alternative definition of *WFSX* which does not rely on 3–valued logic but on 2–valued logic alone. Finally we present some example of logic programs for representing default (or defeasible) rules.

This definition of *WFSX* is also an important consequence of the established relationship. It allows for viewing *WFSX* as a partial 2–valued semantics, where undefined literal are those that can neither be proven true nor false, i.e. those that their truth in a 2–valued logic is *"unknown"*.

Parts of this chapter have been published in [Pereira *et al.*, 1992c].

6.1 The language of defaults

First we review the language of propositional defaults, and some known default logics.

Definition 6.1.1 (Default rule) *A propositional default d is a triple*

$$d = \langle p(d),\ j(d),\ c(d) \rangle$$

where $p(d)$ and $c(d)$ are propositional formulas and $j(d)$ is a finite subset of propositional formulas. $p(d)$ (resp. $j(d)$, resp. $c(d)$) is called the prerequisite *(resp.* justification*, resp.* consequence*) of default d. The default d is also denoted by*

$$\frac{p(d) \ :\ j(d)}{c(d)}$$

Definition 6.1.2 (Default theory) *A default theory Δ is a pair (D, W) where W is a set of propositional formulas and D is a set of default rules.*

As remarked above the definition of the semantics of default theories is herein defined only for a restricted language, though powerful enough to map extended logic programs. Accordingly we define:

Definition 6.1.3 (Restricted default theory) *A restricted default rule is a default rule*

$$\frac{p(d) \ :\ j(d)}{c(d)}$$

where $p(d)$, $j(d)$, and $c(d)$ are literals.

A restricted default theory Δ is a pair (D, W) where W is a set of literals and D is a set of restricted default rules.

Next we review, for the case of propositional defaults, some known default theory semantics. We start by reviewing Reiter's classical default logic [Reiter, 1980]. Then we review (partly following [Baral and Subrahmanian, 1991]) the well-founded [Baral and Subrahmanian, 1991] and stationary [Przymusinska and Przymusinski, 1993] default logics, which correspond respectively to the well founded [Gelder *et al.*, 1991] and stationary semantics [Przymusinski, 1990a] of (nonextended) logic programs.

6.1.1 Reiter's default semantics

To every default theory Δ Reiter associates the operator Γ_Δ, acting on sets of objective literals called contexts:

Definition 6.1.4 (The Γ_Δ operator) *Let $\Delta = (D, W)$, be a propositional default theory and let E be any set of objective literals, called a context. $\Gamma_\Delta(E)$ is the smallest context which:*

1. *contains W;*
2. *is closed under all derivation rules of the form $\frac{p(d)}{c(d)}$, where $\frac{p(d) \,:\, j(d)}{c(d)} \in D$ and $\neg f \notin E$, for every $f \in j(d)$.*

Intuitively, $\Gamma_\Delta(E)$ represents all objective literals *derivable* from W plus E, closed under all default rules whose justifications are consistent with E.

Definition 6.1.5 (Reiter's default extensions) *A context E is an extension of a default theory Δ iff:*

$$E = \Gamma_\Delta(E)$$

The cautious default semantics of Δ is the context consisting of all objective literals which belong to all extensions of Δ.

As argued in [Przymusinska and Przymusinski, 1993], default extensions can be viewed as *rational sets of conclusions* deducible from Δ.

One problem of Reiter's default logic is that it may have multiple extensions and in that case the cautious default semantics is not an extension. If one views extensions as the only rational sets of conclusion then, surprisingly, the (cautious) semantics is not itself one such set.

Example 6.1 Consider the default theory Δ :

$$\left(\left\{ \frac{c \;:\; \neg a}{b}, \frac{c \;:\; \neg b}{a} \right\}, \{c\} \right)$$

which has two extensions:

$$\begin{aligned} E_1 &= \{a, \neg b, c\} \\ E_2 &= \{b, \neg a, c\} \end{aligned}$$

The cautious default semantics is $\{c\}$, itself not an extension, and thus, according to Reiter's semantics, is not a rational set of conclusions.

Another problem is that, in cases where a definite meaning is expected, no extensions exist (and thus no meaning is given).

Example 6.2 The default theory:

$$\left(\left\{ \frac{:\; \neg q}{q} \right\}, \{p\} \right)$$

has no extensions. However p is a fact, and we would expect it to be true.

6.1.2 Well–founded and stationary default semantics for normal logic programs

Here we review two approaches which relate normal logic programs with default theories, and resolve the above mentioned issues of Reiter's default logic.

Baral and Subrahmanian [Baral and Subrahmanian, 1991] introduced the well founded semantics for (propositional) default theories giving a meaning to default theories with multiple extensions. Furthermore, the semantics is defined for all theories, identifying a single extension for each.

Let $\Delta = (D, W)$ be a default theory, and let $\Gamma_\Delta(E)$ be as above. Since $\Gamma_\Delta(E)$ is antimonotonic $\Gamma^2_\Delta(E)$ is monotonic [Baral and Subrahmanian, 1991], and thus has a least fixpoint[1].

Definition 6.1.6 (Well founded semantics)

- *A formula F is* true *in a default theory Δ with respect to the well–founded semantics iff $F \in lfp(\Gamma^2)$.*
- *F is* false *in Δ w.r.t. the well founded semantics iff $F \notin gfp(\Gamma^2)$.*
- *Otherwise F is said to be* unknown *(or* undefined*).*

This semantics is defined for all theories and is equivalent to the Well Founded Model semantics [Gelder *et al.*, 1991] of normal logic programs.

More recently [Przymusinska and Przymusinski, 1993], Przymusinska and Przymusinski generalized this work by introducing the notion of stationary default extensions[2].

Definition 6.1.7 (Stationary extension) *Given a default theory Δ, E is a stationary default extension iff:*

- $E = \Gamma^2_\Delta(E)$
- $E \subseteq \Gamma_\Delta(E)$

Definition 6.1.8 (Stationary default semantics) *Let E be a stationary extension of a default theory Δ.*

- *A formula L is true in E iff $L \in E$.*
- *A formula L is false in E iff $L \notin \Gamma_\Delta(E)$.*
- *Otherwise L is said to be* undetermined *(or* undefined*).*

This semantics has been shown equivalent to stationary semantics of normal logic programs.

Remark 6.1.1 *Note that every default theory has at least one stationary default extension. The least stationary default extension always exists, and corresponds to the well founded semantics for default theories above. Moreover, the least stationary default extension can be computed by iterating the monotonic operator Γ^2_Δ.*

Example 6.3 Consider the default theory of example 6.2. We have $\Gamma_\Delta(\{p\}) = \{p, q\}$ and $\Gamma^2_\Delta(\{p\}) = \{p\}$. p is true in the theory Δ.

[1] Least wrt set inclusion in contexts.

[2] In [Przymusinska and Przymusinski, 1993] the work of [Baral and Subrahmanian, 1991] is also generalized to deal with nonpropositional default theories.

6.2 Some principles required of default theories

Next we argue about some principles a default theory semantics should enjoy, and relate it to logic programs extended with explicit negation, where the said principles are also considered desirable.

Property 6.2.1 (Uniqueness of minimal extension) *We say that a default theory has the uniqueness of minimal extension property if when it has an extension it has a minimal one.*

It is well known that Reiter's default theories do not comply with this principle, which plays an important rôle, specially if we consider the so called *cautious version* of a default semantics [McDermott, 1982]:

Example 6.4 Consider the default theory

$$\left\{\frac{republican(X) \ : \ \neg pacifist(X)}{\neg pacifist(X)}, \frac{quaker(X) \ : \ pacifist(X)}{pacifist(X)}\right\}$$

$$\{republican(nixon), quaker(nixon)\}$$

where Reiter's semantics identifies two extensions:

$$\begin{array}{rcl} E_1 & = & \{ \quad pacifist(nixon),\ republican(nixon),\ quaker(nixon) \ \} \\ E_2 & = & \{ \ \neg pacifist(nixon),\ republican(nixon),\ quaker(nixon) \ \} \end{array}$$

Thus the cautious Reiter's semantics is

$$\{republican(nixon), quaker(nixon)\}$$

As noted in [Przymusinska and Przymusinski, 1993], if we view an extension as a rational set of conclusions, it is strange that the cautious semantics itself does not constitute one such set.

By obeying the uniqueness of minimal extension property, a default semantics avoids this problem. Moreover, this property also eases finding iterative algorithms to compute the cautious version of a default semantics.

Definition 6.2.1 (Union of theories) *The union of two default theories*

$$\Delta_1 = (D_1, W_1) \quad \text{and} \quad \Delta_2 = (D_2, W_2)$$

with languages $L(\Delta_1)$ and $L(\Delta_2)$ is the theory:

$$\Delta = \Delta_1 \cup \Delta_2 = (D_1 \cup D_2, W_1 \cup W_2)$$

with language $L(\Delta) = L(\Delta_1) \cup L(\Delta_2)$.

Example 6.5 Consider the two default theories:

$$\begin{array}{rcl} \Delta_1 & = & \left(\left\{\frac{\ : \ \neg a}{\neg a}, \frac{\ : \ a}{a}\right\}, \{\}\right) \\ \Delta_2 & = & \left(\left\{\frac{\ : \ b}{b}\right\}, \{\}\right) \end{array}$$

Classical default theory, well–founded semantics, and stationary semantics all identify $\{b\}$ as the single extension of Δ_2.

Since the languages of the two theories are disjoint, one would expect their union to include b in all its extensions. However, both the well founded semantics as well as the least stationary semantics give the value undefined to b in the union theory; therefore they are not modular. There is an objectionable interaction among the default rules of both theories when put together. In this case, classical default theory is modular but has two extensions: $\{\neg a, b\}$ and $\{a, b\}$, failing to give a unique minimal extension to the union.

Property 6.2.2 (Modularity) *Let Δ_1, Δ_2 be two default theories with consistent extensions such that $L(\Delta_1) \cap L(\Delta_2) = \{\}$ and let $\Delta = \Delta_1 \cup \Delta_2$, with extensions $E^i_{\Delta_1}$ $E^j_{\Delta_2}$ and E^k_{Δ}. A semantics for default theories is* modular *iff:*

$$\forall_A(\forall_i A \in E^i_{\Delta_1} \Rightarrow \forall_k A \in E^k_{\Delta})$$
$$\forall_A(\forall_j A \in E^j_{\Delta_2} \Rightarrow \forall_k A \in E^k_{\Delta})$$

Informally, a default theory semantics is modular if any theory resulting from the union of two consistent theories with disjoint language contains the consequences of each of the theories alone.

Proposition 6.2.1 *Reiter's default logic is modular.*

Proof: Since a modular theory must be consistent by definition, the disjoint alphabets of two theories can never interact. $\diamondsuit$

Consider now the following examples:

Example 6.6 The default theory

$$\left(\left\{ d_1 = \frac{: \neg b}{a}, d_2 = \frac{: \neg a}{b} \right\}, \{\} \right).$$

has two classical extensions, $\{a\}$ and $\{b\}$. Stationary default semantics has one more extension, namely $\{\}$.

Example 6.7 Let (D, W) be:

$$\left(\left\{ d_1 = \frac{: \neg b}{a}, d_2 = \frac{: \neg a}{b} \right\}, \{\neg a\} \right).$$

The only classical extension is $\{\neg a, b\}$. In the least stationary extension, $E = \Gamma^2_{\Delta}(E) = \{\neg a\}$, $j(d_2) \in E$ but $c(d_2) \notin E$.

Definition 6.2.2 (Applicability of defaults) *Given an extension E:*

- *a default d is* applicable *in E iff $p(d) \subseteq E$ and $\neg j(d) \cap E = \{\}$*
- *an applicable default d is* applied in E *iff $c(d) \in E$*

In classical default semantics every applicable default is applied. This prevents the uniqueness of a minimal extension. In example 6.6, because one default is always applied, one can never have a single minimal extension. In [Baral and Subrahmanian, 1991, Przymusinska and Przymusinski, 1991, Przymusinska and Przymusinski, 1993], in order to guarantee a unique minimal extension, it becomes possible to apply or not an applicable default. However, this abandons the notion of maximality of application of defaults of classical default theory. But, in example 6.7, we argue that at least rule d_2 should be applied.

We want to retain the principle of uniqueness of minimal extension coupled with a notion of maximality of application of defaults we call enforcedness.

Property 6.2.3 (Enforcedness) *Given a theory Δ with extension E, a default d is* enforceable in E *iff $p(d) \in E$ and $j(d) \subseteq E$. An extension* is enforced *if all enforceable defaults in D are applied.*

We argue that, whenever E is an extension, if a default is enforceable then it must be applied. Note that an enforceable default is always applicable.

Another way of viewing enforcedness is that if d is an enforceable default, and E is an extension, then the default rule d must be understood as an inference rule $p(d), j(d) \rightarrow c(d)$ and so $c(d) \in E$ must hold.

The well founded semantics and stationary semantics both sanction minimal extensions where enforceable defaults are not applied, viz. example 6.7. However, in this example they still allow an enforced extension $\{b, \neg a\}$. This is not the case in general:

Example 6.8 Let $(D, W) = \left(\left\{ \frac{: \neg b}{c}, \frac{: \neg a}{b}, \frac{: \neg a}{a} \right\}, \{\neg b\} \right)$. The only stationary extension is $\{\neg b\}$, which is not enforced.

Based on this notion of enforcedness (first presented in [Pereira *et al.*, 1992c]), in [Przymusinska and Przymusinski, 1993], Przymusinska and Przymusinki defined saturated default theories:

Definition 6.2.3 (Saturated default theory) *A default theory $\Delta = (D, W)$ is saturated iff for every default rule*

$$\frac{p(d) \ : \ j(d)}{c(d)} \in D$$

if $p(d) \in W$ and $j(d) \subseteq W$, then $c(d) \in W$.

For this class of default theories they prove that both stationary and well founded default semantics comply with enforcedness. However considering only saturated default theories is a severe restriction since it requires a kind of closure in the theory W.

6.3 Ω-default theory

Next we introduce a default theory semantics which is modular and enforced for every (restricted) default theory. Moreover, when it is defined it has a unique minimal extension.

In the sequel, whenever unambigous, we refer to restricted default rules and theories, simply as default rules and theories.

In order to relate default theories to extended logic programs, we must provide a modular semantics for default theories, except if they are contradictory, as in the example below:

Example 6.9 In the default theory:

$$\left(\left\{ \frac{:}{\neg a}, \frac{:}{a} \right\}, \{\} \right)$$

its two default rules with empty prerequesites and justifications should always be applied, which clearly enforces a contradiction. Note that this would also be the case if the default theory were to be written as $(\{\}, \{a, \neg a\})$.

Consider now example 6.5, that alerted us about nonmodularity in stationary default semantics, where $D = \left\{ \frac{: \neg a}{\neg a}, \frac{: a}{a}, \frac{: b}{b} \right\}$, and $\{\}$ is the least stationary extension.

This result is obtained because $\Gamma_\Delta(\{\})$, by having a and $\neg a$ forces, via the deductive closure, $\neg b$ (and all the other literals) to belong to it. This implies the non–applicability of the third default in the second iteration. For that not to happen one should inhibit $\neg b$ from belonging to $\Gamma_\Delta(\{\})$, which can be done by preventing, in the deductive closure in Γ, the explosion of

conclusions in presence of inconsistency[3]. This is one reason why [Baral and Subrahmanian, 1991]'s use of Γ^2_Δ does not extend to programs with explicit negation.

In our restricted language this is not problematic, because as formulae are just literals, the inhibition of that principle can simply be made by renaming negative literals[4], without side–effects.

Definition 6.3.1 ($\Gamma'_\Delta(E)$) *Let $\Delta = (D, W)$ be a propositional default theory and E a context. Let E' be the smallest set of atoms which:*

1. *contains W';*
2. *is closed under all derivation rules of the form $\frac{p(d)'}{c(d)'}$, such that $\frac{p(d) \,:\, j(d)}{c(d)} \in D$ and $\neg f \notin E$, for every $f \in j(d)'$, and $f \notin E$ for every $\neg\_f \in j(d)'$.*

where W' (resp. $p(d)'$, $j(d)'$, and $c(d)'$) is obtained from W (resp. $p(d)$, $j(d)$, and $c(d)$) by replacing in it every negative literal $\neg A$ by a new atom $\neg\_A$.

$\Gamma'_\Delta(E)$ is obtained from E' by replacing every atom of the form $\neg\_A$ by $\neg A$.

Reconsider now example 6.7, that showed that stationary default extensions are not always enforced. The non–enforced extension is (the least extension) $E = \Gamma^2(E) = \{\neg a\}$, where $\Gamma(E) = \{\neg a, a, b\}$. The semantics obtained is that $\neg a$ is true and a is undefined.

To avoid this counterintuitive result we want to ensure that, for an extension E :

$$\forall d \in D \quad \neg c(d) \in E \Rightarrow c(d) \notin \Gamma(E),$$

i.e. if $\neg c(d)$ is true then $c(d)$ is false[5].

It is easily recognized that this condition is satisfied by seminormal default theories: if $\neg c(d)$ belongs to an extension then any seminormal rule with conclusion $c(d)$ cannot be applied. This principle is exploited in the default semantics.

Definition 6.3.2 (Seminormal version of a default theory) *Given a default theory Δ, its seminormal version[6] Δ^s is obtained by replacing each default rule $d = \frac{p(d) \,:\, j(d)}{c(d)}$ in Δ by the default rule*

$$d^s = \frac{p(d) \;:\; j(d), c(d)}{c(d)}.$$

Definition 6.3.3 (Ω_Δ operator) *For a theory Δ we define:*

$$\Omega_\Delta(E) = \Gamma'_\Delta(\Gamma'_{\Delta^s}(E)).$$

Definition 6.3.4 (Ω–extension) *Let Δ be a default theory. E is an extension iff:*

- $E = \Omega_\Delta(E)$

[3]By the explosion of conclusions we mean the principle *"Ex Contradictione Sequitur Quot Libet"* (From a contradiction everything follows), which is a property of the deductive closure in classical logic. Wagner [Wagner, 1991b] argues against this principle.

[4]In general, this can be achieved by introducing a paraconsistent deductive closure operator. An example of such an operator and its use in default logic can be found in [Pearce, 1992b]. In [Pearce, 1992a], D. Pearce applies this operator to the default logic semantics defined here, and extends it to general default theories.

[5]Note the similarity with the coherence principle.

[6]In Reiter's formalization a default is seminormal if it is of the form

$$\frac{p(d) \;:\; j(d) \wedge c(d)}{c(d)}.$$

Since we are only considering ground versions of the defaults the definitions are equivalent.

- $E \subseteq \Gamma'_{\Delta^s}(E)$

Based on Ω–extensions we define the semantics of a default theory.

Definition 6.3.5 (Ω–default semantics) *Let Δ be a default theory, E an extension of Δ, and L a literal.*

- *L is* true *w.r.t. extension E iff $L \in E$*
- *L is* false *w.r.t. extension E iff $L \notin \Gamma'_{\Delta^s}(E)$*
- *Otherwise L is* undefined

The Ω–default semantics of Δ is determined by the set of all Ω–extensions of Δ.
The cautious Ω–default semantics of Δ is determined by the least Ω–extensions of Δ[7].

Like in [Przymusinska and Przymusinski, 1993], we also require that each extension E be a subset of $\Gamma'_{\Delta^s}(E)$[8]. By not doing so (i.e. considering as extensions all the fixpoints of Ω), the semantics would allow for an objective literal to be both true and false in some extensions.

Example 6.10 For the default theory

$$\Delta = \left(\left\{ \frac{: \neg a}{a}, \frac{: \neg b}{b}, \frac{a : \neg a}{c}, \frac{b : \neg b}{c} \right\}, \{\} \right)$$

there are four fixpoints of Ω_Δ :

$$\begin{array}{lllllll} E_1 & = & \{\} & & \Gamma'_{\Delta^s}(E_1) & = & \{a,b,c\} \\ E_2 & = & \{a,c\} & & \Gamma'_{\Delta^s}(E_2) & = & \{b,c\} \\ E_3 & = & \{b,c\} & & \Gamma'_{\Delta^s}(E_3) & = & \{a,c\} \\ E_4 & = & \{a,b,c\} & & \Gamma'_{\Delta^s}(E_4) & = & \{\} \end{array}$$

Only E_1 is an extension, and thus it determines the Ω–default semantics of Δ..

Note how, for instance, $a \in E_2$ and $a \notin \Gamma'_{\Delta^s}(E_2)$. Thus, if E_2 were to be considered as an extension a would be both true and false in E_2.

Moreover, intuitively no extension should contain c, since for each rule with conclusion c, the prerequisites are incompatible with the justification. In E_2 c is true because a being true satisfies the prerequisites, and a being false satisfies the justifications.

This definition of extension guarantees that no pair of contradictory literals belongs to E.

Proposition 6.3.1 *If E is a Ω–extension of a default theory Δ then:*

$$\nexists L \mid \{L, \neg L\} \subseteq E.$$

Proof: Assume the contrary, i.e. $\exists L \mid \{L, \neg L\} \subseteq E$ and E is an extension. By seminormality, $L \notin \Gamma'_{\Delta^s}(E)$ and $\neg L \notin \Gamma'_{\Delta^s}(E)$. Thus $E \not\subseteq \Gamma'_{\Delta^s}(E)$, and so is not an extension. $\diamondsuit$

Example 6.11 Consider the default theory

$$\Delta = \left(\left\{ \frac{: \neg c}{c}, \frac{: \neg b}{a}, \frac{: \neg a}{b}, \frac{:}{\neg a} \right\}, \{\} \right).$$

Its only extension is $\{\neg a, b\}$.

In fact:

$$\begin{array}{rcll} \Gamma'_{\Delta^s}(\{\neg a, b\}) & = & \{c, b, \neg a\} & \text{and} \\ \Gamma'_{\Delta}(\{c, b, \neg a\}) & = & \{\neg a, b\} & \end{array}$$

Thus $\neg a$ and b are true, c is undefined, and a and $\neg b$ are false.

[7] The existence of a least extension is guaranteed by theorem 6.3.4 below.

[8] In [Przymusinska and Przymusinski, 1993] the requirement is wrt $\Gamma_\Delta(E)$ instead of wrt $\Gamma'_{\Delta^s}(E)$.

It is easy to see that some theories may have no Ω–extension.

Example 6.12 The theory $\Delta = (\{\frac{\cdot}{a}, \frac{\cdot}{\neg a}\}, \{\})$ has no Ω–extension.

Definition 6.3.6 (Contradictory theory) *A default theory Δ is contradictory iff it has no Ω–extension.*

In order to guarantee the existence of a least extension we prove:

Theorem 6.3.1 (Ω is monotonic) *If Δ is a noncontradictory theory then Ω_Δ is monotonic.*

Proof: We begin by stating two lemmas:

Lemma 6.3.2 *Let $\Delta = (D, W)$ be a noncontradictory default theory, and*

$$\Delta' = \left(D \cup \left\{ \frac{\cdot}{L} \mid L \in W \right\}, \{\} \right).$$

E is an Ω–extension of Δ iff is an Ω–extension of Δ'.

Proof: It is easy to see that every Ω–extension of Δ and of Δ' contains W. Thus for each Ω–extension of one of the theories the set of rules in D applied is the same as in the other theory. $\diamondsuit$

Lemma 6.3.3 *If Δ is a noncontradictory default theory then Γ'_Δ is antimonotonic.*

Proof: Without loss of generality (cf. lemma 6.3.2 above) we consider

$$\Delta = (D, \{\}).$$

First we define two transformations over sets of objective literals, and one over default theories.

- A^- is a set of atoms obtained from a set of objective literals A by replacing every negative literal $\neg L$ by the new atom $\neg\_L$.
- A^+ is a set of objective literals obtained from a set of atom A by replacing every atom of the form $\neg\_L$ by the objective literal $\neg L$.
- Δ^{--} is the default theory obtained from $\Delta = (D, W)$ by replacing in D every occurence of a negative literal $\neg A$ by the new atom $\neg\_A$.

Clearly, the first two transformations are monotonic, i.e.:

$$A \subseteq B \Rightarrow A^+ \subseteq B^+$$

$$A \subseteq B \Rightarrow A^- \subseteq B^-$$

Directly from the definition of Γ'_Δ, and given that we are assuming $W = \{\}$, and Δ is noncontradictory:

$$\Gamma'_\Delta(A) = (\Gamma_{\Delta^{--}}(A^-))^+ \quad (*)$$

Now we prove that:

$$A \subseteq B \Rightarrow \Gamma'_\Delta(B) \subseteq \Gamma'_\Delta(A)$$

By monotonicity of A^- :

$$A \subseteq B \Rightarrow A^- \subseteq B^-$$

Given that Γ is antimonotonic for any default theory:

$$A^- \subseteq B^- \Rightarrow \Gamma_{\Delta^{--}}(B^-) \subseteq \Gamma_{\Delta^{--}}(A^-)$$

By monotonicity of A^+ :

$$\Gamma_{\Delta^{--}}(B^-) \subseteq \Gamma_{\Delta^{--}}(A^-) \Rightarrow (\Gamma_{\Delta^{--}}(B^-))^+ \subseteq (\Gamma_{\Delta^{--}}(A^-))^+$$

By the result of $(*)$:

$$(\Gamma_{\Delta^{--}}(B^-))^+ \subseteq (\Gamma_{\Delta^{--}}(A^-))^+ \Rightarrow \Gamma'_\Delta(B) \subseteq \Gamma'_\Delta(A)$$

i.e. Γ'_Δ is antimonotonic. $\diamondsuit$

Since Ω_Δ is the composition of two antimonotonic operators, it is monotonic. $\diamondsuit$

Definition 6.3.7 (Iterative construction) *To obtain a constructive definition for the least (in the set inclusion order sense) Ω-extension of a theory we define the following transfinite sequence $\{E_\alpha\}$:*

$$\begin{array}{rcl} E_0 & = & \{\} \\ E_{\alpha+1} & = & \Omega(E_\alpha) \\ E_\delta & = & \bigcup\{E_\alpha \mid \alpha < \delta\} \quad \textit{for limit ordinal } \delta \end{array}$$

By theorem 6.3.1, and the Knaster–Tarski theorem [Tarski, 1955], there must exist a smallest ordinal λ for the sequence above, such that E_λ is the smallest fixpoint of Ω. If E_λ is a Ω–extension then it is the smallest one. Otherwise, by the proposition below, there are no Ω–extensions for the theory.

Proposition 6.3.2 *If the least fixpoint E of Ω_Δ is not a Ω–extension of Δ then Δ has no Ω–extensions.*

Proof: We prove that if there exists an extension E^* of Ω_Δ, then the least fixpoint of Ω_Δ is an extension.

Assume that such an E^* exists. Given that, by hypothesis, E is the least fixpoint of Ω_Δ, $E \subseteq E^*$.

On the assumption that E^* is an extension, Δ is noncontradictory and, by lemma 6.3.3, Γ'_{Δ^s} is antimonotonic. Thus:

$$E \subseteq E^* \quad \Rightarrow \Gamma'_{\Delta^s}(E^*) \subseteq \Gamma'_{\Delta^s}(E)$$

Since, by hypothesis, E^* is an extension, $E^* \subseteq \Gamma'_{\Delta^s}(E^*)$. Thus:

$$E^* \subseteq \Gamma'_{\Delta^s}(E^*) \subseteq \Gamma'_{\Delta^s}(E)$$

Again using the fact that $E \subseteq E^*$:

$$E \subseteq E^* \subseteq \Gamma'_{\Delta^s}(E^*) \subseteq \Gamma'_{\Delta^s}(E)$$

Thus $E \subseteq \Gamma'_{\Delta^s}(E)$, and so E is an extension of Ω_Δ. $\diamondsuit$

Example 6.13 Consider the default theory Δ of example 6.11. In order to obtain the least (and only) extension of Δ we build the sequence:

$$\begin{array}{rcllllll} E_0 & = & \{\} & & & & & \\ E_1 & = & \Gamma'_\Delta(\Gamma'_{\Delta^s}(\{\})) & = & \Gamma'_\Delta(\{c,a,b,\neg a\}) & = & \{\neg a\} & \\ E_2 & = & \Gamma'_\Delta(\Gamma'_{\Delta^s}(\{\neg a\})) & = & \Gamma'_\Delta(\{c,b,\neg a\}) & = & \{\neg a, b\} & \\ E_3 & = & \Gamma'_\Delta(\Gamma'_{\Delta^s}(\{\neg a, b\})) & = & \Gamma'_\Delta(\{c,b,\neg a\}) & = & \{\neg a, b\} & = \quad E_2 \end{array}$$

Because $E_2 \subseteq \Gamma'_{\Delta^s}(E_2)$, it is the least Ω–extension of Δ.

Example 6.14 Let $\Delta = (\ \{\frac{:}{a}, \frac{:}{\neg a}\}, \{\})$. Let us build the sequence:

$$\begin{array}{lllllllll} E_0 & = & \{\} & & & & & & \\ E_1 & = & \Gamma'_\Delta(\Gamma'_{\Delta^s}(\{\})) & = & \Gamma'_\Delta(\{a, \neg a\}) & = & \{a, \neg a\} & & \\ E_2 & = & \Gamma'_\Delta(\Gamma'_{\Delta^s}(\{a, \neg a\})) & = & \Gamma'_\Delta(\{\}) & = & \{a, \neg a\} & = & E_1 \end{array}$$

Since $E_1 \not\subseteq \Gamma'_{\Delta^s}(E_1)$, Δ has no Ω–extensions.

We will now prove that this new default semantics satisfies all the principles required above (section 6.2).

Theorem 6.3.4 (Uniqueness of minimal extension) *If Δ has an extension then there is one least extension E.*

Proof: Trivial, given that Ω_Δ is monotonic for noncontradictory program. $\diamondsuit$

Theorem 6.3.5 (Enforcedness) *If E is a Ω–extension then E is enforced.*

Proof: Without loss of generality (cf. lemma 6.3.2 above) we consider

$$\Delta = (D, \{\}).$$

We want to prove that for any default rule d :

$$p(d) \in E \text{ and } j(d) \subseteq E \Rightarrow c(d) \in E$$

If $j(d) \subseteq E$ then, by seminormality, no rule with a conclusion $\neg f$, such that $f \in j(d)$, is applicable in $\Gamma'_{\Delta^s}(E)$. So, given that we are assuming $W = \{\}$ for theory Δ :

$$\text{for all literals } f \text{ in } j(d),\ \neg f \notin \Gamma'_{\Delta^s}(E).$$

Thus the default d is applicable in $\Gamma'_\Delta\Gamma'_{\Delta^s}(E)$, i.e., by definition of Γ, $\Gamma'_\Delta\Gamma'_{\Delta^s}(E)$ must be closed under the derivation rule $\frac{p(d)}{c(d)}$.

Given that E is an Ω–extension:

$$p(d) \in E \Rightarrow p(d) \in \Gamma'_\Delta\Gamma'_{\Delta^s}(E)$$

and because $\Gamma'_\Delta\Gamma'_{\Delta^s}(E)$ must be closed under that derivation rule:

$$c(d) \in \Gamma'_\Delta\Gamma'_{\Delta^s}(E)$$

Again because E is an extension, if $c(d) \in \Gamma'_\Delta\Gamma'_{\Delta^s}(E)$ then $c(d) \in E$. $\diamondsuit$

Corollary 6.3.1 *If E is an Ω–extension of Δ then for any $d = \frac{:}{c(d)} \in \Delta$, $c(d) \in E$.*

Proof: Follows directly from enforcedness for true prerequisites and justifications. $\diamondsuit$

Theorem 6.3.6 (Modularity) *Let L_{Δ_1} and L_{Δ_2} be the languages of two default theories. If $L_{\Delta_1} \cap L_{\Delta_2} = \{\}$ then, for any corresponding extensions E_1 and E_2, there always exists an extension E of $\Delta = \Delta_1 \cup \Delta_2$ such that $E = E_1 \cup E_2$.*

Proof: Since the languages are disjoint, the rules of Δ_1 and Δ_2 do not interact on that count. Additionally, since there is no explosion of conclusions in the presence of inconsistency, one can never obtain the whole set of literals as a result of a contradictory Γ'_{Δ^s}, and hence they do not interact on that count either. $\diamondsuit$

6.4 Comparison with Reiter's semantics

Comparing this semantics for defaults theories with Reiter's, we prove that for restricted default theories (cf. definition 6.1.3) the former is a generalization of the latter, in the sense that whenever Reiter's semantics (Γ–extension) gives a meaning to a theory (i.e. the theory has at least one Γ–extension), Ω semantics provides one too.

Moreover, whenever both semantics give meaning to a theory Ω semantics is sound w.r.t. the intersection of all Γ–extensions. Thus we provide a monotonic fixpoint operator for computing a subset of the intersection of all Γ–extensions. For that purpose we begin by stating and proving:

Theorem 6.4.1 *Consider a theory Δ such that Ω–semantics is defined. Then every Γ–extension is a Ω–extension.*

Proof: First two lemmas:

Lemma 6.4.2 *If E is consistent and $E = \Gamma_\Delta(E)$ then $E = \Gamma'_{\Delta^s}(E)$.*

Proof: By definition of Γ_Δ,

$$E = \Gamma_\Delta(E) \Longrightarrow (\forall_{d \in D} \;\; p(d) \in E \;\; \wedge \;\; \neg j(d) \cap E = \{\} \Rightarrow c(d) \in E).$$

Thus, since E is consistent:

$$\forall_{d \in D}\; p(d) \in E \wedge \neg j(d) \cap E = \{\} \wedge \neg c(d) \cap E = \{\} \Rightarrow c(d) \in E$$

and so, by definition of Γ'_{Δ^s}, it follows easily that $E = \Gamma'_{\Delta^s}(E)$. $\diamondsuit$

Lemma 6.4.3 *If E is consistent and $E = \Gamma_\Delta(E)$ then $E = \Gamma'_\Delta(E)$.*

Proof: Similar to the one of lemma 6.4.2. $\diamondsuit$

Now we prove that for an E such that

$$E = \Gamma_\Delta(E)$$

$E = \Omega_\Delta(E)$ holds.

By definition,

$$\Omega_\Delta(E) = \Gamma'_\Delta(\Gamma'_{\Delta^s}(E)).$$

By lemma 6.4.2,

$$\Omega_\Delta(E) = \Gamma'_\Delta(E).$$

And by lemma 6.4.3,

$$\Gamma'_\Delta(E) = E.$$

For E to be a Ω–extension one more condition must hold:

$$E \subseteq \Gamma'_{\Delta^s}(E).$$

It is easy to recognize given the hypothesis

$$E = \Gamma_\Delta(E).$$

$\diamondsuit$

The next two results follow directly from the above theorem.

Theorem 6.4.4 (Generalization of Reiter's semantics) *If a theory Δ has at least one Γ–extension, it has at least one Ω–extension.*

Theorem 6.4.5 (Soundness wrt to Reiter's semantics) *If a theory Δ has a Γ–extension, whenever L belongs to the least Ω–extension it also belongs to the intersection of all Γ–extensions.*

It is interesting to note that any other combination of the Γ–like operators that are used to define Ω (i.e. the operators: $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$, $\Gamma'^2_{\Delta^s}$, and Γ'^2_{Δ}) also give semantics that are sound wrt Reiter's, but which are not as close to the latter as the semantics defined by Ω. By "not as close" we mean that its least fixpoints are subsets of the intersection of all Reiter's extensions, that are smaller (wrt set inclusion) than the least fixpoint of Ω. Thus we say that Ω is the best approximation of Reiter's default semantics, when compared to the others.

Proposition 6.4.1 *Let Δ be a noncontradictory default theory. Then:*

1. $lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}) \subseteq lfp(\Gamma'^2_{\Delta^s})$
2. $lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}) \subseteq lfp(\Gamma'^2_{\Delta})$
3. $lfp(\Gamma'^2_{\Delta^s}) \subseteq lfp(\Omega)$
4. $lfp(\Gamma'^2_{\Delta}) \subseteq lfp(\Omega)$

Proof: In appendix. $\diamondsuit$

6.5 Comparison with stationary default semantics

We now draw some brief comparisons with stationary extensions [Przymusinska and Przymusinski, 1993]. It is not the case that every stationary extension is a Ω–extension since, as noted above, non–modular or non–enforced stationary extensions are not Ω–extensions. As shown in the example below, it is also not the case that every Ω–extension is a stationary extension.

Example 6.15 Let Δ be:

$$\left(\left\{ \frac{:\ \neg b}{c}, \frac{:\ \neg a}{b}, \frac{:\ \neg a}{a}, \frac{:}{\neg b} \right\}, \{\} \right)$$

The only Ω–extension of Δ is $\{c, \neg b\}$. This is not a stationary extension.

As stated above, for saturated default theories stationary semantics complies with enforcedness. However, even for this class of theories, the two semantics might not coincide. This is because in general stationary default extensions are not modular.

Example 6.16 The default theory of example 6.5 is saturated and has a non–modular stationary extension.

However, in a large class of cases these semantics coincide. In particular:

Proposition 6.5.1 *If for every default $d = \frac{p(d)\ :\ j(d)}{c(d)}$ $c(d)$ is a positive literal then Ω coincides with Γ^2_{Δ}.*

Proof: For such theories $\Gamma'_{\Delta^s} = \Gamma'_{\Delta} = \Gamma_{\Delta}$. Thus $\Gamma'_{\Delta}\Gamma'_{\Delta^s} = \Gamma^2_{\Delta}$. $\diamondsuit$

6.6 Relation between the semantics of default theories and logic programs with explicit negation

Here we state the equivalence of Ω–extensions and partial stable models of extended logic programs as defined in chapter 4. For the sake of brevity proofs are in appendix C.

Definition 6.6.1 (Program corresponding to a default theory) *Let $\Delta = (D, \{\})$ be a default theory. We say an extended logic program P corresponds to Δ iff:*

- *For every default of the form:*

$$\frac{\{a_1, \ldots, a_n\} \ : \ \{b_1, \ldots, b_m\}}{c} \in \Delta$$

there exists a rule

$$c \leftarrow a_1, \ldots, a_n, not\ \neg b_1, \ldots, not\ \neg b_m \in P$$

where $\neg b_j$ denotes the $\neg$–complement of b_j.

- *No rules other than these belong to P.*

Definition 6.6.2 (Interpretation corresponding to a context) *An interpretation I of a program P corresponds to a default context E of the corresponding default theory T iff for every objective literal L of P (and literal L of T):*

- $I(L) = 1$ *iff* $L \in E$ *and* $L \in \Gamma'_{\Delta^s}(E)$.
- $I(L) = \frac{1}{2}$ *iff* $L \notin E$ *and* $L \in \Gamma'_{\Delta^s}(E)$.
- $I(L) = 0$ *iff* $L \notin E$ *and* $L \notin \Gamma'_{\Delta^s}(E)$.

The main theorem relating both semantics is now presented:

Theorem 6.6.1 (Correspondence) *Let $\Delta = (D, \{\})$ be a default theory corresponding to program P. E is a Ω-extension of Δ iff the interpretation I corresponding to E is a partial stable model of P.*

According to this theorem we can say that explicit negation is nothing but classical negation in (restricted) default theories, and vice–versa. As Ω default semantics is a generalization of Γ default semantics (cf. theorems 6.4.4 and 6.4.5), and since answer-sets semantics corresponds to Γ default semantics [Gelfond and Lifschitz, 1990], it turns out that answer-sets semantics (and hence the semantics defined in [Wagner, 1991a]) is a special case of *WFSX*. Other properties of Ω-extensions can also be translated into properties of models of extended logic programs, e.g. modularity, uniqueness of minimal extension, etc.

On the other hand, with this theorem one can rely on the top–down procedures of logic programming to compute default extensions. In particular, in accordance with theorem 6.4.5, the top–down procedures for *WFSX* can be used as sound top–down procedures for Reiter's default logic.

Example 6.17 Consider program P :

$$\begin{array}{rcl} c & \leftarrow & not\ c \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ \neg a & \leftarrow & \end{array}$$

The corresponding default theory is

$$\Delta = \left(\left\{ \frac{: \neg c}{c}, \frac{: \neg b}{a}, \frac{: \neg a}{b}, \frac{:}{\neg a} \right\}, \{\} \right).$$

As calculated in example 6.11, the only Ω–extension of Δ is $E = \{\neg a, b\}$ and $\Gamma'_{\Delta^s}(E) = \{\neg a, b, c\}$. The PSM corresponding to this extension is

$$M = \{\neg a, not\ a, b, not\ \neg b, not\ \neg c\}^9.$$

It is easy to verify that M is the only PSM of P.

6.7 A definition of *WFSX* based on Γ

In [Gelfond and Lifschitz, 1990], it is proven that, with the above correspondences between programs and default theories, and between interpretations and default contexts, Reiter's Γ operator for defaults is equivalent to the Gelfond–Lifschitz (GL) Γ operator for extended logic programs (cf. definition 2.2.1). Thus, the above relationship between *WFSX* and Ω extensions directly suggests an alternative definition of *WFSX*.

Based on this relationship, and on the fact that the GL Γ operator is not based on 2–valued logic, in this section we present an alternative definition of *WFSX* not relying in a 3–valued logic, but rather on a partial 2–valued logic.

We begin by defining in logic programs the notion corresponding to seminormality in default theories.

Definition 6.7.1 (Seminormal version of a program) *The seminormal version of a program P is the program P_s obtained from P by adding to the (possibly empty) Body of each rule:*

$$L \leftarrow Body$$

the default literal not $\neg L$, where $\neg L$ is the complement of L wrt explicit negation.

For short, when P is understood from context, we use $\Gamma(S)$ to denote $\Gamma_P(S)$, and $\Gamma_s(S)$ to denote $\Gamma_{P_s}(S)$.

Theorem 6.7.1 (Partial stable models) *Let P be an extended logic program.*

$$M = T \cup not\ F$$

is a partial stable model of P iff:

$$(1) \quad T = \Gamma\Gamma_s T$$
$$(2) \quad T \subseteq \Gamma_s T$$

Moreover $F = \{L \mid L \notin \Gamma_s T\}$, and members of $\Gamma_s T$ not in T are undefined *in M.*
In the sequel we refer to T as the generator of M.

Proof: Follows directly from theorem 6.6.1. $\diamondsuit$

Note that in these alternative definitions each PSM is completely determined by the objective literals true in it.

Theorem 6.7.2 (Well–founded model) *Let P be a noncontradictory program.*
$M = T \cup not\ F$ is the well–founded model of P iff T is the least fixpoint of $\Gamma\Gamma_s$ and generates M.

[9] Note that c is undefined in M.

Thus the WFM can be obtained by iterating $\Gamma\Gamma_s$ from the empty set. If a fixpoint S is reached, then it contains objective literals true in the WFM. False literals in it are the ones compatible with $\Gamma_s S$, i.e. those literals not in $\Gamma_s S$. It is also possible to define an iterative construction of false literals in the WFM, and determine instead true literal from false ones.

The next proposition helps us build one such iterative construction.

Proposition 6.7.1 *Let P be a noncontradictory program. Then:*

$$\Gamma_s(lfp(\Gamma\Gamma_s)) = gfp(\Gamma_s\Gamma)$$

Proof: First we prove that $\Gamma_s(lfp(\Gamma\Gamma_s))$ is a fixpoint of $\Gamma_s\Gamma$. By definition:

$$lfp(\Gamma\Gamma_s) = \Gamma\Gamma_s(lfp(\Gamma\Gamma_s))$$

Thus:

$$\Gamma_s(lfp(\Gamma\Gamma_s)) = \Gamma_s(\Gamma\Gamma_s(lfp(\Gamma\Gamma_s)))$$

By associativity of function compositions:

$$\Gamma_s(lfp(\Gamma\Gamma_s)) = \Gamma_s\Gamma(\Gamma_s(lfp(\Gamma\Gamma_s)))$$

i.e. $\Gamma_s(lfp(\Gamma\Gamma_s))$ is a fixpoint of $\Gamma_s\Gamma$.

Now let S be a fixpoint of $\Gamma_s\Gamma$. We have to prove that:

$$S \subseteq \Gamma_s(lfp(\Gamma\Gamma_s))$$

To that proof, we begin by showing that $lfp(\Gamma\Gamma_s) \subseteq \Gamma S$
Given that $\Gamma\Gamma_s$ is monotonic, there exists a smallest ordinal λ such that:

$$lfp(\Gamma\Gamma_s) = \Gamma\Gamma_s^{\uparrow\lambda}\{\}$$

We now prove by transfinite induction that for any ordinal α

$$\Gamma\Gamma_s^{\uparrow\alpha}\{\} \subseteq \Gamma S$$

- *For limit ordinals:* If $\alpha = 0$[10] then trivially:

$$\{\} \subseteq \Gamma S$$

For limit ordinal δ, suppose that for all $\alpha < \delta$

$$\Gamma\Gamma_s^{\uparrow\alpha}\{\} \subseteq \Gamma S$$

Then clearly

$$\bigcup\left\{\Gamma\Gamma_s^{\uparrow\alpha}\{\} \mid \alpha < \delta\right\} \subseteq \Gamma S$$

i.e.

$$\Gamma\Gamma_s^{\uparrow\delta}\{\} \subseteq \Gamma S$$

[10]For the sake of clarity, here and in all proofs by transfinite induction in this report, we show the special case of limit ordinal 0.

- *Induction step:* Assume that for some ordinal i

$$\Gamma\Gamma_s^{\uparrow i}\{\} \subseteq \Gamma S$$

Then, given that $\Gamma\Gamma_s$ is monotonic:

$$\Gamma\Gamma_s(\Gamma\Gamma_s^{\uparrow i})\{\} \subseteq \Gamma\Gamma_s(\Gamma S)$$

By associativity of function compositions, this inequality is equivalent to:

$$\Gamma\Gamma_s^{\uparrow i+1}\{\} \subseteq \Gamma(\Gamma_s\Gamma S)$$

Given that by hypothesis S is a fixpoint of $\Gamma_s\Gamma$:

$$\Gamma\Gamma_s^{\uparrow i+1}\{\} \subseteq \Gamma S$$

At this point we've proven that $lfp(\Gamma\Gamma_s) \subseteq \Gamma S$. From this result, and given that Γ_s is antimonotonic, it follows that:

$$\Gamma_s\Gamma S \subseteq \Gamma_s(lfp(\Gamma\Gamma_s))$$

Again because by hypothesis S is a fixpoint of $\Gamma_s\Gamma$:

$$S \subseteq \Gamma_s(lfp(\Gamma\Gamma_s))$$

$\diamondsuit$

We now define two (monotonic) operators: one which given a set of true objective literals, determines additional true objective literals; another which given a set of false objective literals determines additional false objective literals.

Definition 6.7.2 *For a program P define:*

$$\begin{array}{rcl} \mathcal{T}(S) & = & \Gamma\Gamma_s(S) \\ \mathcal{F}(S) & = & \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} - S) \end{array}$$

where $\mathcal{H}$ denotes the Herbrand base of P.

Theorem 6.7.3 *For any noncontradictory program, both $\mathcal{T}$ and $\mathcal{F}$ are monotonic.*

Proof: The proof of monotonicity of $\mathcal{T}$ is trivial given that of Ω for defaults (theorem 6.3.1), and that a program is noncontradictory iff the corresponding default theory is also noncontradictory. This last results follows directly from theorem 6.6.1.

Similarly to the proof of theorem 6.3.1, one can prove that $\Gamma_s\Gamma$ is also monotonic. So:

$$\begin{array}{rcrcll} A \subseteq B & \Rightarrow & \mathcal{H} - B & \subseteq & \mathcal{H} - A & \Rightarrow \\ & \Rightarrow & \Gamma_s\Gamma(\mathcal{H} - B) & \subseteq & \Gamma_s\Gamma(\mathcal{H} - A) & \Rightarrow \\ & \Rightarrow & \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} - A) & \subseteq & \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} - B) & \Rightarrow \\ & \Rightarrow & \mathcal{F}(A) & \subseteq & \mathcal{F}(B) & \end{array}$$

i.e. $\mathcal{F}$ is monotonic. $\diamondsuit$

Theorem 6.7.4 *Let P be a noncontradictory program. Then:*

$$WFM(P) = lfp(\mathcal{T}) \cup not\ lfp(\mathcal{F})$$

Proof: We begin with the lemma:

Lemma 6.7.5 *For any noncontradictory program:*

$$lfp(\mathcal{F}) = \mathcal{H} - gfp(\Gamma_s\Gamma)$$

Proof: We begin by proving by transfinite induction that:

$$\mathcal{F}^{\uparrow\alpha}\{\} = \mathcal{H} - (\Gamma_s\Gamma)^{\downarrow\alpha}\mathcal{H}$$

- *For limit ordinals:* Since $\mathcal{F}^{\uparrow 0}\{\} = \{\}$, and $\mathcal{H} - (\Gamma_s\Gamma)^{\downarrow 0}\mathcal{H} = \mathcal{H} - \mathcal{H}$, the condition holds for $\alpha = 0$.

  For a limit ordinal δ, suppose that for all $\alpha < \delta$:

  $$\mathcal{F}^{\uparrow\alpha}\{\} = \mathcal{H} - (\Gamma_s\Gamma)^{\downarrow\alpha}\mathcal{H}$$

  Then, clearly:

  $$\bigcup\left\{\mathcal{F}^{\uparrow\alpha}\{\} \mid \alpha < \delta\right\} = \mathcal{H} - \bigcap\left\{(\Gamma_s\Gamma)^{\downarrow\alpha}\mathcal{H} \mid \alpha < \delta\right\}$$

  i.e.

  $$\mathcal{F}^{\uparrow\delta}\{\} = \mathcal{H} - (\Gamma_s\Gamma)^{\downarrow\delta}\mathcal{H}$$

- *Induction step:* Assume that for some ordinal i

  $$\mathcal{F}^{\uparrow i}\{\} = \mathcal{H} - (\Gamma_s\Gamma)^{\downarrow i}\mathcal{H}$$

  Then:

  $$\mathcal{F}^{\uparrow i+1}\{\} = \mathcal{F}(\mathcal{F}^{\uparrow i}\{\}) = \mathcal{F}(\mathcal{H} - (\Gamma_s\Gamma)^{\downarrow i}\mathcal{H})$$

  Applying the definition of $\mathcal{F}$:

  $$\mathcal{F}^{\uparrow i+1}\{\} = \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} - (\mathcal{H} - (\Gamma_s\Gamma)^{\downarrow i}\mathcal{H}))$$

  Given that for any two sets A and B, $B - (B - A) = B \cap A$:

  $$\mathcal{F}^{\uparrow i+1}\{\} = \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} \cap (\Gamma_s\Gamma)^{\downarrow i}\mathcal{H})$$

  Since the result of $\Gamma_s\Gamma$ is a subset of the Herbrand base, i.e. for any S, $\mathcal{H} \supseteq \Gamma_s\Gamma S$:

  $$\mathcal{F}^{\uparrow i+1}\{\} = \mathcal{H} - \Gamma_s\Gamma((\Gamma_s\Gamma)^{\downarrow i}\mathcal{H}) = \mathcal{H} - (\Gamma_s\Gamma)^{\downarrow i+1}\mathcal{H}$$

Given this result, the proof follows directly from the iterative construction of least and gretaest fixpoints of monotonic operators. ♢

According to this lemma and proposition 6.7.1:

$$lfp(\mathcal{F}) = \mathcal{H} - \Gamma_s(lfp(\Gamma\Gamma_s))$$

From theorem 6.7.2:

$$WFM(P) = lfp(\mathcal{T}) \cup not\ (\mathcal{H} - \Gamma_s(lfp(\Gamma\Gamma_s)))$$

♢

Example 6.18 Consider the program P :

$$\begin{array}{rcl} c & \leftarrow & b, not\ c \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ \neg a & & \end{array}$$

Next we show two alternative ways of computing the WFM.

1. Start from an empty set of true objective literals, and iterate consecutively, in order to get more objective literals true, until a fixpoint is reached:

$$\begin{array}{lcl} T_0 & = & \{\} \\ T_1 & = & \Gamma\Gamma_s\{\} = \Gamma\{c,a,b,\neg a\} = \{\neg a\} \\ T_2 & = & \Gamma\Gamma_s\{\neg a\} = \Gamma\{c,b,\neg a\} = \{b,\neg a\} \\ T_3 & = & \Gamma\Gamma_s\{b,\neg a\} = \Gamma\{c,b,\neg a\} = \{b,\neg a\} \end{array}$$

Then:

$$\begin{array}{lcl} WFM & = & T_3 \cup not\ (\mathcal{H} - \Gamma_s T_3) \\ & = & \{b,\neg a\} \cup not\ (\mathcal{H} - \{c,b,\neg a\}) \\ & = & \{b,\neg a\} \cup \{not\ a, not\ \neg b, not\ \neg c\} \end{array}$$

2. Start from an empty set of false objective literals and iterate consecutively, in order to get more objective literals false, until a fixpoint is reached:

$$\begin{array}{lcl} F_0 & = & \{\} \\ F_1 & = & \mathcal{H} - \Gamma_s\Gamma(\mathcal{H} - \{\}) = \mathcal{H} - \Gamma_s\{\neg a\} \\ & = & \mathcal{H} - \{c,b,\neg a\} = \{a,\neg b,\neg c\} \\ F_2 & = & \mathcal{H} - \Gamma_s\Gamma\{c,b,\neg a\} = \mathcal{H} - \Gamma_s\{b,\neg a\} \\ & = & \mathcal{H} - \{c,b,\neg a\} = \{a,\neg b,\neg c\} \end{array}$$

Then:

$$\begin{array}{lcl} WFM & = & \Gamma(\mathcal{H} - F_2) \cup not\ F_2 \\ & = & \Gamma\{c,b,\neg a\} \cup \{not\ a, not\ \neg b, not\ \neg c\} \\ & = & \{b,\neg a\} \cup \{not\ a, not\ \neg b, not\ \neg c\} \end{array}$$

6.8 Logic programming for default reasoning

The purpose of this section is to show some examples of how extended logic programming, given its close relationship with defaults, can be used to formalize problems of default (or defeasible) reasoning. It is not our purpose here to give a formal methodology of how to write defeasible rule in extended programs. For that the reader is referred to [Pereira *et al.*, 1991g, Pereira *et al.*, 1992f, Pereira *et al.*, 1993b, Aparício, 1993]

6.8.1 Hierarchical taxonomies

Here we illustrate how to represent taxonomies with extended logic programs. In this representation we wish to express general absolute (i.e. non–defeasible) rules, defeasible rules, exceptions to defeasible rules, as well as exceptions to exceptions, explicitly making preferences among defeasible rules. We also show how to express preference for one defeasible rule over another whenever they conflict. In taxonomic hierarchies we wish to express that in the presence of contradictory defeasible rules we prefer the one with most specific[11] information (e.g. for a penguin, which is a bird, we want to conclude that it doesn't fly).

The statements about the domain are:

(1) Mammals are animals.
(2) Bats are mammals.
(3) Birds are animals.
(4) Penguins are birds.
(5) Dead animals are animals.
(6) Normally animals don't fly.
(7) Normally bats fly.
(8) Normally birds fly.
(9) Normally penguins don't fly.
(10) Normally dead animals don't fly.

[11] In [Nute, 1986], the author suggests using this notion of more specific information to resolve conflicts between contradictory defeasible rules.

and the following elements:

(11) Pluto is a mammal.
(12) Tweety is a bird.
(13) Joe is a penguin.
(14) Dracula is a bat.
(15) Dracula is a dead animal.

depicted as in fig. 6.1, and the preferences:

(16) Dead bats do not fly though bats do.
(17) Dead birds do not fly though birds do.
(18) Dracula is an exception to the above preferences.

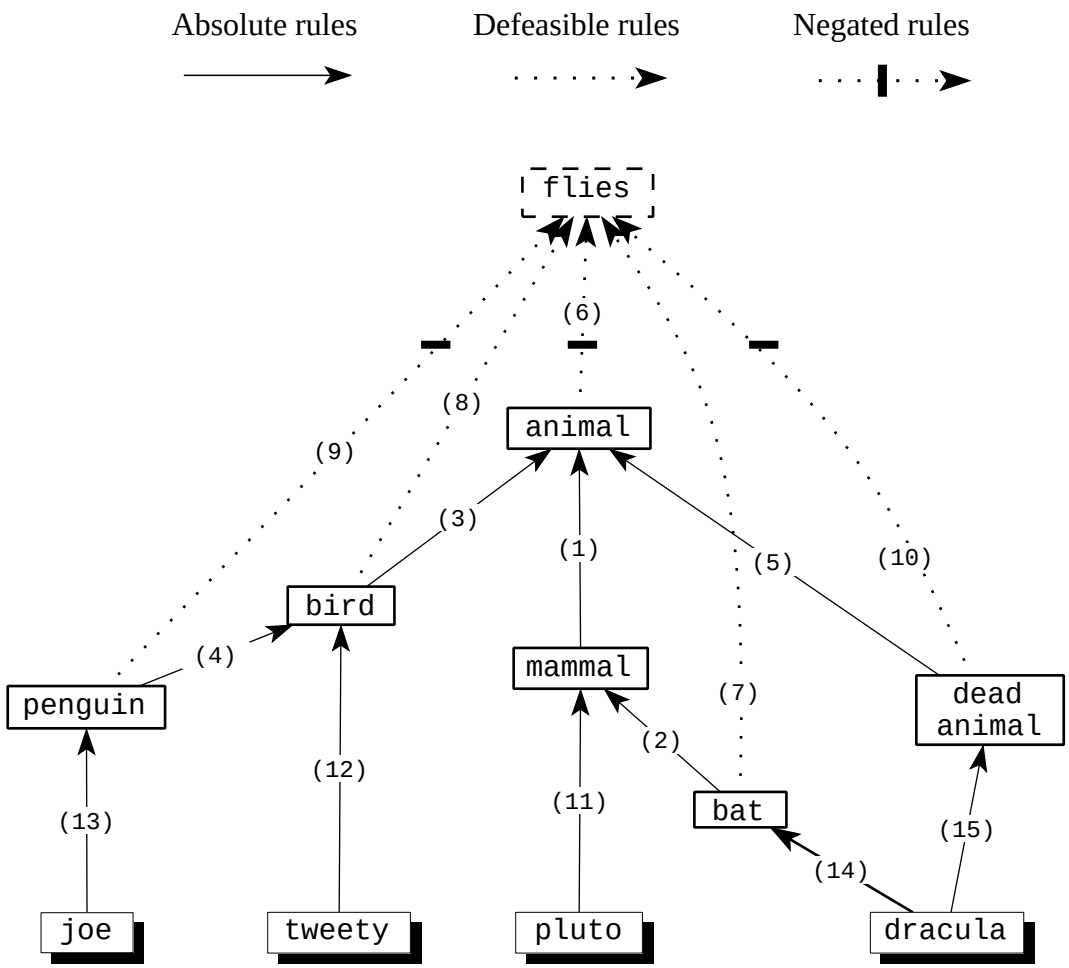

Figure 6.1: A hierarchical taxonomy

The above hierarchy can be represented by the program:

$$
\begin{aligned}
animal(X) &\leftarrow mammal(X) && (1)\\
mammal(X) &\leftarrow bat(X) && (2)\\
animal(X) &\leftarrow bird(X) && (3)\\
bird(X) &\leftarrow penguin(X) && (4)\\
animal(X) &\leftarrow dead\_animal(X) && (5)\\
\neg flies(X) &\leftarrow animal(X), not\ ab_1(X) && (6)\\
flies(X) &\leftarrow bat(X), not\ ab_2(X) && (7)\\
flies(X) &\leftarrow bird(X), not\ ab_3(X) && (8)\\
\neg flies(X) &\leftarrow penguin(X), not\ ab_4(X) && (9)\\
\neg flies(X) &\leftarrow dead\_animal(X), not\ ab_5(X) && (10)\\
mammal(pluto) & && (11)\\
bird(tweety) & && (12)\\
penguin(joe) & && (13)\\
bat(dracula) & && (14)\\
dead\_animal(dracula) & && (15)
\end{aligned}
$$

with the implicit hierarchical preference rules (not shown in fig. 6.1):

$$
\begin{aligned}
ab_1(X) &\leftarrow bat(X), not\ ab_2(X)\\
ab_1(X) &\leftarrow bird(X), not\ ab_3(X)\\
ab_3(X) &\leftarrow penguin(X), not\ ab_4(X)
\end{aligned}
$$

and the explicit problem statement preferences:

$$
\begin{aligned}
ab_2(X) &\leftarrow dead\_animal(X), bat(X), not\ ab_5(X) && (16)\\
ab_3(X) &\leftarrow dead\_animal(X), bird(X), not\ ab_5(X) && (17)\\
ab_5(dracula) & && (18)
\end{aligned}
$$

As expected, this program has exactly one partial stable model (coinciding with its well founded model), no choice being possible and everything being defined in the hierarchy. The model is given by the table in figure 6.2 where $\surd$ means that the predicate (in the row entry) is true about the element (in the column entry), e.g. $penguin(joe)$ holds in the model.

Thus pluto doesn't fly, and isn't an exception to any of the rules; tweety flies because it's a bird and an exception to the "animals don't fly" rule; joe doesn't fly because it's a penguin and an exception to the "birds fly" rule.

Although dracula is a dead animal, which by default don't fly (cf. rule (10)) it is also considered an exception to this very same rule. Furthermore rule (16) saying that "dead bats normally do not fly" is also exceptioned by dracula and thus the "bats fly" rule applies and dracula flies.

Note that preferences rules must be present in order to prevent contradiction to arise, thus preference rules play the rôle of removing contradictions arising in the initial specification of the problem.

6.8.2 Possible worlds

In hierarchies, as seen, everything is defined, leaving no choices available. This is not the case for general defeasible reasoning problems. In this section we show an example (the so called "Nixon diamond") of how to represent defeasible reasoning problems in *WFSX* and interpret the results.

Consider the statments:

- *Normally quakers are pacifists.*

individ. / predicat.	joe	dracula	pluto	tweety
dead_animal	*not* , *not* ¬	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
bat	*not* , *not* ¬	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
penguin	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
mammal	*not* , *not* ¬	√ , *not* ¬	√ , *not* ¬	*not* , *not* ¬
bird	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬	√ , *not* ¬
animal	√ , *not* ¬	√ , *not* ¬	√ , *not* ¬	√ , *not* ¬
ab_4	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
ab_2	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
ab_3	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
ab_1	*not* , *not* ¬	√ , *not* ¬	*not* , *not* ¬	√ , *not* ¬
ab_5	*not* , *not* ¬	√ , *not* ¬	*not* , *not* ¬	*not* , *not* ¬
flies	¬ , *not*	√ , *not* ¬	¬ , *not*	√ , *not* ¬

Figure 6.2: The well founded model of the hierarchy

- *Normally republicans are hawks.*
- *Pacifists are non hawks.*
- *Hawks are non pacifists.*
- *Nixon is a quaker and a republican.*
- *There are other quakers.*
- *There are other republicans.*

rendered by the program:

$$
\begin{array}{rcll}
 & & \text{Normally quakers are pacifists} & \\
pacifist(X) & \leftarrow & quaker(X), quaker\_pacifist(X), & \\
 & & not\ \neg pacifist(X). & \\
quaker\_pacifist(X) & \leftarrow & not\ \neg quaker\_pacifist(X). &
\end{array}
$$

$$
\begin{array}{rcll}
 & & \text{Normally republicans are hawks} & \\
hawk(X) & \leftarrow & republican(X), republican\_hawk(X), & \\
 & & not\ \neg hawk(X). & \\
republican\_hawk(X) & \leftarrow & not\ \neg republican\_hawk(X). &
\end{array}
$$

$$
\begin{array}{rcll}
\neg hawk(X) & \leftarrow & pacifist(X). & \text{Pacifists are non hawks} \\
\neg pacifist(X) & \leftarrow & hawk(X). & \text{Hawks are non pacifists}
\end{array}
$$

$$
\begin{array}{rl}
quaker(nixon). & \text{Nixon is quaker.} \\
republican(nixon). & \text{Nixon is republican.} \\
quaker(other\_quaker). & \text{There are other quakers.} \\
republican(other\_republican). & \text{There are other republicans}
\end{array}
$$

where *other_quaker* and *other_republican* can be envisaged as Skolem constants.

Here there is no preference defined between the rules nor between their conclusions. So for nixon we want to be able to hypothesize him to be a pacifist or a hawk. Let us have a closer look at the partial stable models of this program. The WFM is[12]:

$$\{qua(n), rep(n), qp(n), rh(n),$$
$$qua(o\_q), qp(o\_q), rh(o\_q), pac(o\_q), \neg hawk(o\_q), not\ \neg pac(o\_q), not\ hawk(o\_q),$$
$$rep(o\_r), qp(o\_r), rh(o\_r), hawk(o\_r), \neg pac(o\_r), not\ \neg hawk(o\_q), not\ pac(o\_q)\}$$

For nixon, as expected, it is unknown whether he is a pacifist or a hawk. Nevertheless we have quaker_pacifist(nixon) and republican_hawk(nixon), so that both rules applied. This is not a strange result since, as for defaults, rules are maximally applicable, and nixon does not consist an exception to the rules through to their conclusions. Of course, for other quakers and other republicans everything is defined.

Since we have unknown literals not present in the WF Model we might have other partial stable models. In this case the other PSMs are[13]:

$$PSM_1 = WFM \cup \{pac(n), \neg hawk(n)\}$$
$$PSM_2 = WFM \cup \{\neg pac(n), hawk(n)\}$$

These remaining PSMs can be seen as possible extended world views in which some consistent choices of belief have been made. PSM_1 represents a world view where nixon is a pacifist and a non hawk, and PSM_2 represents a world view where nixon is a hawk and a non pacifist.

[12]Where *qua* stands for *quaker*, *rep* for *republican*, *pac* for *pacifist*, *qp* for *quaker_pacifist*, *rh* for *republican_hawk*, *n* for *nixon*, *o_q* for *other_quaker* and *o_r* for *other_republican*.

[13]As these two models are 2-valued we don't show the additional *not* literals, which are implicit.

Chapter 7

WFSX as hypotheses abduction

Approaches to nonmonotonic reasoning semantics clash on two major intuitions: scepticism and credulity [Touretzky *et al.*, 1987]. In normal logic programming the credulous approach includes semantics such as stable models [Gelfond and Lifschitz, 1988] and preferred extensions [Dung, 1991], while the well–founded semantics [Gelder *et al.*, 1991] is the sole representative of scepticism [Dung, 1991].

In extended logic programming, while generalizations of stable models semantics are clearly credulous in their approach, no semantics whatsoever has attempted to seriously explore the sceptical approach. A closer look at some of the works generalizing well-founded semantics [Dung and Ruamviboonsuk, 1991, Przymusinski, 1990a, Przymusinski, 1991b, Sakama, 1992] shows these generalizations to be rather technical in nature, where the different techniques introduced to formally characterize the well–founded semantics of normal logic programs are slightly modified in some way to become applicable to the more general case.

In this chapter we characterize a spectrum of more or less sceptical and credulous semantics for extended logic programs, and determine the position of *WFSX* in this respect.

We do so by means of a coherent, flexible, unifying, and intuition appealing framework for the study of explicit negation in logic programs, based on the notion of admissible scenaria. This framework extends the approach originally proposed in [Dung, 1991] for normal logic programs.

The basic idea of the framework is to consider default literals as abducibles, i.e. they must be hypothesized. This idea was first proposed in [Eshghi and Kowalski, 1989], and in [Dung, 1991] it was further explored in order to capture stable models [Gelfond and Lifschitz, 1988] and the well–founded semantics [Gelder *et al.*, 1991] of normal programs. There, an hypothesis is acceptable iff there is no evidence to the contrary: roughly no set of hypotheses derives its complement[1]. Semantics are then defined by adding to a program sets of acceptable hypotheses, according to additional specific choice criteria. Depending on the chosen criteria, more sceptical or credulous semantics are obtained.

In trying to extend these notions to extended logic programs, a new kind of hypotheses appears – mandatory hypotheses.

Example 7.1 Consider a program containing the rules:

$$\begin{array}{rcl} tryBus & \leftarrow & not\ driversSrike \\ \neg driversStrike & & \end{array}$$

[1] In [Bondarenko *et al.*, 1993] the authors develop an assumption–based argumentation framework for logic programming where a variety of alternative of evidence to the contrary notions are studied. In our approach the notion of evidence to the contrary is kept fixed.

advising to plan a trip by bus if it can be assumed the bus drivers are not on strike, and stating bus drivers are not on strike. No matter what the rest of the program is (assuming it is consistent on the whole), it is clear that a rational agent assumes the bus drivers are not on strike, and of course he plans his trip by bus.

In this case it is mandatory to assume the hypothesis *not driversSrike*.

Intuitively, an hypothesis *not* L is mandatory if $\neg L$ is a consequence of the program, i.e. if objective literal L is explicitly stated false then the hypothesis that assumes it false must per force be accepted. This amounts to the coherence principle.

In other words, in extended programs default literals can be view as hypotheses, where an objective literal L inhibits the hypothesis *not* L (as in normal programs), and $\neg L$ makes the assumption of hypothesis *not* L imperative.

Moreover, viewing default literals as hypotheses that may or may not be accepted, helps us provide semantics for contradictory programs where contradiction is brought about by such hypotheses[2]. Indeed, if default literals are just hypotheses, and if some of them cause contradiction, then it seems natural not to accept these in order to assing a meaning to a program. Even though there may be no specific evidence to the contrary of an hypothesis, if its adoption leads to a global contradiction then its acceptance is questionable. This is an instance of the *"reductio ad absurdum"* principle.

In this section to begin we define, in a simple way, an ideal sceptical semantics and its well–founded (or grounded) part, in fact an entirely declarative semantics able to handle programs like:

$$\begin{array}{rcl} a & \leftarrow & not\ p \\ \neg a & \leftarrow & not\ q \end{array} \qquad \begin{array}{rcl} b & \leftarrow & not\ r \end{array}$$

and assigning it the semantics $\{b, not\ r\}$.

WFSX cannot deal with such programs because, as neither p nor q have rules, it assumes both *not* p and *not* q without regard to the ensuing contradiction, except as an after–the–fact filter. In our ideal sceptical semantics this program is not contradictory at all.

However, the issue of dealing with such contradictory programs within *WFSX*, and assigning to them a semantics is explored in detail in chapter 8, where we use the framework of this chapter, plus the additional notion of optative hypothesis, as its basis.

One advantage of viewing logic programs as abduction is its close relationship with argumentation system and dialogue games. In [Dung *et al.*, 1992, Kakas *et al.*, 1993], the authors have pointed out the similarities between the ideas of acceptability of hypotheses and evidence to the contrary, and the notions of arguments and attacks of argumentation systems. Based on that they sustain that [Dung, 1991] is in fact an argumentational approach to normal logic programs. In the same way, our approach can be viewed as an argumentational approach to extended logic programs.

The problem of understanding the process of argumentation (or dialogue games) has been addressed by many researchers in different fields [Toulmin, 1958, Birnbaum *et al.*, 1980, McGuire *et al.*, 1981, Hintikka, 1983, Cohen, 1987, Pollock, 1992]. The understanding of the structure and acceptability of arguments is essential for a computer system to be able to engage in exchanges of arguments with other systems.

The ability of viewing extended logic programs as argumentation systems opens the way for its use in formalizing communication among reasoning computing agents in a distributed

[2]Note that these are the cases presented above, where *WFSX* provides no meaning and we argue that it might be natural to provide one.

framework [Nejdl *et al.*, 1993].

A dialogue game is an exchange of arguments between two players where each alternately presents arguments attacking the arguments of the opponent. The player who fails to present counterarguments looses the game. As shown in [Dung, 1992a, Dung, 1993a, Dung, 1993b] a game theoretical semantics for logic programming can be defined by interpreting programs as schemas for forming arguments, where a literal can be concluded if it is supported by acceptable arguments constructed according to the rules of the program:

Example 7.2 Consider program P :

$$\begin{array}{rcll} \neg fly(X) & \leftarrow & animal(X), not\ ab\_a(X) & animal(tweety) \\ ab\_a(X) & \leftarrow & bird(X), not\ ab\_b(X) & bird(tweety) \\ ab\_b(X) & \leftarrow & penguin(X) & penguin(tweety) \end{array}$$

P can be viewed as the rules for constructing the arguments:

1. Tweety does not fly since it is an animal and animals normally do not fly.
2. Tweety is an abnormal animal since it is a bird and normally birds are abnormal animals with respect to flying.
3. Tweety is an abnormal bird since it is a penguin and penguins are abnormal birds with respect to flying.

A dialogue game to determine whether or not tweety flies proceeds as follows:

- Player 1 presents argument 1 supporting the conclusion that tweety cannot fly. His argument is based on the assumption that animals normally do not fly.
- In the next move player 2 presents argument 2 which *"attacks"* argument 1 by defeating the assumption made by the latter. His argument is based on the assumption that normally birds are abnormal animals.
- Then player 1 presents argument 3 *"counterattacking"* the argument of player 2.
- As player 2 cannot find any argument *counterattacking* the argument of player 1, he looses the game and gives up his claims.

In the framework we present in this chapter, hypotheses can be viewed as arguments, that may or may not be accepted, in the same way arguments may or may not be winning ones. An argument is acceptable if every attack against it can be counterattacked by it. As we point out below, this is tantamount to the acceptance of hypotheses, where an hypothesis is acceptable in the context of other hypotheses if every set of hypotheses that constitutes evidence to its contrary is in turn defeated by the context where it is accepted. To make this clearer we explain, for the program of example 7.2, why $not\ ab\_a(tweety)$ is acceptable:

The hypotheses $not\ ab\_a(tweety)$ is acceptable because the only evidence to the contrary, i.e. to $ab\_a(tweety)$, is the hypothesis $not\ ab\_b(tweety)$, and this evidence is defeated by $not\ ab\_a(tweety)$: in the context where this assumption is made true in the program $ab\_b(tweety)$ follows as a consequence.

A detailed study of logic programming as dialogue games and argumentation systems is not in the scope of this work. However, the intuitions behind the relationship between the concepts introduced here and those of dialogue games and argumentation systems can be found throughout this chapter.

Parts of this chapter appear in [Alferes *et al.*, 1993].

7.1 Admissible scenaria for extended logic programs

In this section we generalize the notions of scenario and evidence for normal logic programs given in [Dung, 1991], to those extended with explicit negation. They are reminiscent of the notions of scenario and extensions of [Poole, 1988].

In [Dung, 1991, Brogi *et al.*, 1992, Dung *et al.*, 1992] a normal logic program is viewed as an abductive framework where literals of the form *not* L (NAF–hypotheses) are considered as new atoms, say *not*_L, and are abducibles, i.e. they must be hypothesized. The set of all ground NAF–hypotheses is *not* $\mathcal{H}$, where $\mathcal{H}$ denotes the Herbrand base of the program, as usual, and *not* prefixed to a set denotes the set obtained by prefixing *not* to each of its elements[3]. Here we generalize these notions to extended logic programs.

In order to introduce explicit negation we first consider negated objective literals of the form $\neg A$ as new symbols (as in [Gelfond and Lifschitz, 1988]). The Herbrand base is now extended to the set of all such objective literals. Of course, this is not enough to correctly treat explicit negation. Relations among $\neg A$, A, and *not* A, must be established, as per the definitions below.

Definition 7.1.1 (Scenario) *A scenario of an extended logic program P is the Horn theory $P \cup H$, where $H \subseteq$ not $\mathcal{H}$.*

For scenaria we define a derivability operator in a straightforward way, given that every scenario is a Horn theory:

Definition 7.1.2 ($\vdash$ operator) *Let P be an extended logic program and H a set of NAF–hypotheses.*

P' is the Horn theory obtained from P by replacing:

- *every objective literal of the form $\neg L$ by the atom $\neg\_L$*
- *every default literal of the form not L by the atom not_L*
- *every default literal of the form not $\neg L$ by the atom not_$\neg_L$*

where $\neg\_L$, not_L, and not_$\neg_L$ are new atoms not appearing in P.
A set H' is obtained from H using the same replacement rules.

By definition $P' \cup H'$ is a Horn theory, and so it has a least model M.
We define $\vdash$ in the following way (where A is any atom of P):

$$\begin{array}{lll} P \cup H \vdash A & \textit{iff} & A \in M \\ P \cup H \vdash \neg A & \textit{iff} & \neg\_A \in M \\ P \cup H \vdash \textit{not } A & \textit{iff} & \textit{not}\_A \in M \\ P \cup H \vdash \textit{not } \neg A & \textit{iff} & \textit{not}\_\neg\_A \in M \end{array}$$

In argumentation systems a scenario can be viewed as a possible set of arguments. In particular the arguments corresponding to a scenario $P \cup H$ are those engendered by the hypotheses in H.

When introducing explicit negation into logic programs one has to reconsider the notion of NAF–hypotheses, or simply hypotheses. As the designation "explicit negation" suggests, when a scenario $P \cup H$ entails $\neg A$ it is *explicitly* stating that A is false in that scenario. Thus the hypothesis *not* A is enforced in the scenario, and cannot optionally be held independently. This is the *"coherence principle"*, which relates both negations.

[3] In [Brogi *et al.*, 1992] the authors dub these programs open positive ones. Positive because all negated literals are transformed into new atoms, and open because the program can be completed with additional information, i.e. default literals can be added (or hypothesized) in order to give the program a meaning.

Definition 7.1.3 (Mandatory hypotheses wrt $P \cup H$) *The set of mandatory hypotheses (or mandatories) wrt a scenario $P \cup H$ is:*

$$Mand(H) = \{not\ L \mid P \cup H \cup \{not\ K \leftarrow \neg K \mid K \in \mathcal{H}\} \vdash not\ L\}$$

where L or K is any objective literal, and $\neg K$ denotes the complement of K wrt explicit negation. The extra rules enforce coherence.

Alternatively, the set of mandatory hypotheses wrt $P \cup H$ is the smallest set $Mand(H)$ such that:

$$Mand(H) = \{not\ L \mid P \cup H \cup Mand(H) \vdash \neg L\}.$$

Example 7.3 Consider program P :

$$\begin{array}{rcl} q & \leftarrow & not\ r \\ \neg r & \leftarrow & not\ p \\ \neg p & & \end{array}$$

Then:

$$Mand(\{\}) = \{not\ p, not\ r, not\ \neg q\}.$$

Indeed, the Horn theory:

$$\begin{array}{rcl} q & \leftarrow & not\_r \\ \neg\_r & \leftarrow & not\_p \\ \neg\_p & & \end{array} \qquad \begin{array}{rcl} not\_\neg\_q & \leftarrow & q \\ not\_q & \leftarrow & \neg\_q \\ not\_\neg\_r & \leftarrow & r \\ not\_r & \leftarrow & \neg\_r \end{array} \qquad \begin{array}{rcl} not\_\neg\_p & \leftarrow & p \\ not\_p & \leftarrow & \neg\_p \end{array}$$

derives $\{not\_p, not\_r, not\_\neg\_q\}$ and no more hypotheses.

Example 7.4 Consider the program P :

$$\begin{array}{rcl} b(p) & & \\ \neg m(X,Y) & \leftarrow & b(X) \\ m(p,s) & \leftarrow & not\ m(t,s) \\ m(t,s) & \leftarrow & not\ m(p,s) \\ \neg m(X,X) & & \end{array}$$

obtained from the autoepistemic theory of example 5.17.

The mandatory hypotheses wrt $P \cup \{\}$ are:

- from the last rule, all ground instances of literals of the form $not\ m(X,X)$;
- from the first rule, $not\ \neg b(p)$;
- from the first and second rules $P \vdash \neg m(p,Y)$, and thus ground instances of literals of the form $not\ m(p,Y)$ are mandatories;
- from the above points and the third rule it follows that P and its mandatories derive $m(t,s)$, and so $not\ \neg m(t,s)$ is also mandatory.

Mandatory hypotheses correspond in argumentation systems to arguments that cannot be directly attacked because they are sustained by conclusions. For instance, the fact $\neg fly(tweety)$ in a program states that Tweety does not fly. Since no argument can attack this fact, the argument $not\ fly(tweety)$ is unattackable.

Example 7.5 Consider a program containing the rules:

$$\begin{array}{rcl} newsAboutStrike & \leftarrow & driversStrike \\ \neg driversStrike & & \end{array}$$

stating that newspapers publish news about the strike if the drivers are on strike, and that the bus drivers are definitely not on strike.

For a rational reasoner the second rule should not provide a pretext for newspapers to publish news about a strike by possibly assuming it, since indeed the first rule (or some other) may actually state or conclude the contrary of that assumption.

Note how this is accomplished be using always programs in the canonical form (definition 2.1.1), where any true rule head has the effect of falsifying the body of all rules containing its complement literal wrt explicit negation.

Recall that, within a program in the canonical form, any objective literal L in the body of a rule is to be considered shorthand for the conjunction $L, not\ \neg L$. This allows for technical simplicity in capturing the relation between $\neg L$ and $not\ L$ (cf. justification in the compact version of the modulo operator in chapter 4). Thus, without loss of generality (cf. corollary 10.1.1), and for the sake of technical simplicity, whenever refering to a program in this section we always mean its canonical form. In all examples we expressly use the canonical program.

Definition 7.1.4 (Consistent scenario) *A scenario $P \cup H$ is consistent iff for all objective literals L such that:*

$$P \cup H \cup Mand(H) \vdash L$$

then

$$not\ L \notin H \cup Mand(H)$$

Note that, by the definition of mandatory hypotheses, for every consistent scenario:

$$\text{if} \quad P \cup H \cup Mand(H) \vdash L \quad \text{then} \quad P \cup H \cup Mand(H) \nvdash \neg L.$$

Unlike the case of non–extended logic programs, an extended logic program may in general have no consistent scenaria:

Example 7.6 Program

$$P = \left\{ \begin{array}{rcl} \neg p & & \\ p & \leftarrow & not\ p \end{array} \right\}$$

has no consistent scenario.

Note that $P \cup \{\}$ is not consistent since $Mand(\{\}) = \{not\ p\}$ and $P \cup \{not\ p\} \vdash p$.

A notion of program consistency is needed. Intuitively, a program is consistent iff it has some consistent scenario. Because for a given H, if $P \cup H$ is consistent then $P \cup \{\} \cup Mand(\{\})$ is also consistent, we define:

Definition 7.1.5 (Consistent program) *An extended logic program P is consistent iff*

$$P \cup Mand(\{\})$$

is a consistent scenario.

Inconsistent programs are those that derive a contradiction even without assuming any hypotheses (except, of course, for those for which it is mandatory to do so, i.e. the mandatories). The rôle of the semantics here being to determine sets of hypotheses that can be added to a program without making it inconsistent, and since no set whatsoever is in these conditions for an inconsistent program, no semantics is given it.

By adding to the body of each rule a private default literal *not* L', where L' is a new atom not appearing elsewhere in the program, every program becomes consistent. This operation, similar to the naming device of [Poole, 1988], renders every rule hypothetical because its condition is contingent on the prior acceptance of its private "naming" default literal. Ultimately, inconsistency can thus be always avoided. Semantics that assign meaning to inconsistent programs by considering consistent subsets of its rules can be "simulated" in ours via the naming device.

Thus, from now on, unless otherwise stated, we restrict programs to consistent ones only.

Not every consistent scenario specifies a consensual semantics for a program [Poole, 1988], in the same way that not every set of arguments is a winning set in dialog games. For example [Dung, 1991] the program P :

$$p \leftarrow not\ q$$

has a consistent scenario $P \cup \{not\ p\}$ which fails to give the intuitive meaning of P. It is not consensual to assume *not* p since there is the possibility of p being true (if *not* q is assumed), and $\neg p$ is not explicitly stated (if this were the case then *not* q could not be assumed).

Intuitively, what we wish to express is that a hypothesis can be assumed only if there can be no evidence to the contrary.

Clearly a hypothesis *not* L is only directly contradicted by the objective literal L. Evidence for an objective literal L in a program P is a set of hypotheses which, if assumed in P together with its mandatories, would entail L.

Definition 7.1.6 (Evidence for an objective literal L) *A subset E of not $\mathcal{H}$ is evidence for an objective literal L in a program P iff:*

$$E \supseteq Mand(E) \qquad \text{and} \qquad P \cup E \vdash L^4$$

If P is understood and E is evidence for L we write $E \rightsquigarrow L$.

Note here the similarities between *evidence to the contrary* of an hypothesis and *attack* to an argument.

As in [Dung, 1991] a hypothesis is acceptable wrt a scenario iff there is no evidence to the contrary, i.e. iff all evidence to the contrary is itself defeated by the scenario:

Definition 7.1.7 (Acceptable hypothesis) *A hypothesis not L is acceptable wrt the scenario $P \cup H$ iff:*

$$\forall E : E \rightsquigarrow L \Rightarrow \exists not\ A \in E \mid P \cup H \cup Mand(H) \vdash A,$$

i.e. each evidence for L is defeated by $P \cup H$.

The set of all acceptable hypotheses wrt $P \cup H$ is denoted by $Acc(H)$.

[4] The consistency of $P \cup E$ is not required; e.g. $P \cup \{not\ A\} \vdash A$ is allowed.

This is tantamount to the acceptability of arguments in dialogue games. In the latter an argument is acceptable if it can *counterattack* (i.e. defeat) every *attack* made on it (i.e. every evidence to the contrary).

Example 7.7 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b, not\ c \\ b & \leftarrow & not\ d \\ \neg c & & \end{array}$$

In the scenario $P \cup \{not\ c, not\ d, not\ a\}$:

- $not\ c$ is mandatory because $P \vdash \neg c$;
- $not\ d$ (resp. $not\ \neg a$, $not\ \neg b$) is acceptable because there is no evidence for d (resp. $\neg a$, $\neg b$);
- $not\ a$ is acceptable because any evidence for a must contain

  $$\{not\ b, not\ c\}$$

  and so is defeated by the scenario since

  $$P \cup \{not\ c, not\ d, not\ a\} \cup Mand(\{not\ c, not\ d, not\ a\}) \vdash b$$

For example, $not\ b$ is neither mandatory nor acceptable because, respectively:

$$P \cup \{not\ c, not\ d, not\ a\} \cup Mand(\{not\ c, not\ d, not\ a\}) \nvdash \neg b$$

and $\{not\ d\}$ is an evidence for b not defeated by the scenario, i.e.:

$$P \cup \{not\ d\} \cup Mand(\{not\ d\}) \vdash b$$

and

$$P \cup \{not\ c, not\ d, not\ a\} \cup Mand(\{not\ c, not\ d, not\ a\}) \nvdash d$$

In a consensual semantics we are interested only in admitting consistent scenaria whose hypotheses are either acceptable or mandatory. As the designation "mandatory hypotheses" suggests, any scenario to be considered must include all its mandatory hypotheses:

Definition 7.1.8 (Admissible scenario) *A scenario $P \cup H$ is admissible iff it is consistent and:*

$$Mand(H) \subseteq H \subseteq Mand(H) \cup Acc(H)$$

We must guarantee that by considering only admissible scenaria one does not fail to give semantics to consistent programs, i.e.:

Proposition 7.1.1 *Any consistent program P has at least an admissible scenario.*

Proof: By hypothesis P is consistent and so the scenario $P \cup Mand(\{\})$ is also consistent.

By definition $Mand(H)$ is closed under mandatories, i.e.

$$Mand(H) = Mand(Mand(H))$$

So $P \cup H$, where $H = Mand(\{\})$, is an admissible scenario:

$$Mand(Mand(\{\})) = Mand(\{\}) \subseteq Mand(Mand(\{\})) \cup Acc(Mand(\{\}))$$

♢

The notion of admissible scenario discards all hypotheses which are unacceptable, whatever the semantics of extended logic programs to be defined.

One semantics can be defined as the class of all admissible scenaria, where the meaning of a program is determined, as usual, by the intersection of all such scenaria.

However, since $P \cup Mand(\{\})$ is always the least admissible scenario (cf. proof of proposition 7.1.1), this semantics does not include any non–mandatory hypothesis. Consequently this semantics is equivalent to replacing every *not* L by the corresponding objective literal $\neg L$.

Example 7.8 Let P :

$$\begin{array}{rcl} \neg p & & \\ a & \leftarrow & not\ b \end{array}$$

Its admissible scenaria are:

$$\begin{array}{l} P \cup \{not\ p\} \\ P \cup \{not\ p, not\ \neg a\} \\ P \cup \{not\ p, not\ \neg b\} \\ P \cup \{not\ p, not\ b, not\ \neg a\} \\ P \cup \{not\ p, not\ \neg a, not\ \neg b\} \\ P \cup \{not\ p, not\ b, not\ \neg a, not\ \neg b\} \end{array}$$

the least admissible scenario being the first.

Thus the literals entailed by the semantics of admissible scenaria are $\{\neg p, not\ p\}$. Note *not* b and a are not entailed by this extremely sceptical semantics.

The semantics of admissible scenaria is the most sceptical one for extended logic programs: it contains no hypotheses except for mandatory ones[5]. In order to define more credulous semantics, we define classes of scenaria based on proper subsets of the class of admissible scenaria, as governed by specific choice criteria. Constraining the set of admissible scenaria reduces undefinedness but may restrict the class of programs having a semantics.

In the next sections we define a spectrum of semantics which, by restricting the set of admissible scenaria, are more credulous, but give meaning to narrower classes of programs. *WFSX* turns out to be one of the semantics in that spectrum.

7.2 A sceptical semantics for extended programs

Several proposals, already mentioned above, have been made to generalize well–founded semantics[6] to logic programs with explicit negation, in order to obtain a sceptical semantics for extended logic programs. But a closer look at these works shows these generalizations to be of a rather technical nature, where different techniques introduced to characterize the well–founded semantics of normal logic programs (those without explicit negation) are in someway modified to become applicable to the more general case. So it would not be surprising if tomorrow some new *"sceptical"* semantics for programs with explicit negation were to be presented. So which of them is really "sceptical"? And what is the essential difference between them? How many "sceptical" semantics are we going to have? After all, what makes a semantics "sceptical"? Certainly not just because it is in some way "technically" similar to one or other presentation of the well–founded semantics of Van Gelder et al. [Gelder *et al.*, 1991][7].

[5]This semantics is equivalent to one which only accepts hypotheses if it is explicitly negated in the program that there is evidence to the contrary. Hence it contains only the mandatory literals.

[6]By its nature the representative of scepticism in normal logic programs.

[7]Dung [Dung, 1992b] has shown that stable model semantics can also be viewed as well–founded semantics, since it can be defined a similar way.

It is natural and important to ask the question of what is an ideally sceptical semantics for explicit negation, *i.e. one which would be part of the semantics of every rational reasoner.*

Suppose that $P \cup H$ is this *"ideal"* sceptical semantics. In the previous section, we have introduced and argued that an admissible scenario represents a scenario which is admissible for a rational reasoner. Let one such admissible scenario be $P \cup K$. It is clear that $P \cup K \cup H$ is again admissible since H must be part of this agent's semantics. This leads to an immediate definition of the *"ideal"* or *"idealized"* sceptical semantics.

Definition 7.2.1 (Ideal sceptical semantics) *A set of hypotheses H is called the ideal sceptical semantics, ISS, if it is the greatest set satisfying the condition:*

For each admissible scenario $P \cup K$, $P \cup K \cup H$ is again admissible.

It is clear that if P is consistent then such a set exists, a consequence of the fact that the union of sets satisfying the above condition satisfies it too.

Example 7.9 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ p \\ \neg a & \leftarrow & not\ q \\ c & \leftarrow & not\ r \end{array}$$

The admissible scenaria are (apart from literals $not\ \neg p$, $not\ \neg q$, and $not\ \neg r$, which are irrelevant to this example and are omitted):

$$\begin{array}{ll} P \cup \{\} & \\ P \cup \{not\ \neg c\} & P \cup \{not\ r, not\ \neg c\} \\ P \cup \{not\ \neg c, not\ p, not\ \neg a\} & P \cup \{not\ r, not\ \neg c, not\ p, not\ \neg a\} \\ P \cup \{not\ \neg c, not\ q, not\ a\} & P \cup \{not\ r, not\ \neg c, not\ q, not\ a\} \end{array}$$

It is not difficult to see that the greatest admissible scenario whose union with any other is again admissible is $\{not\ r, not\ \neg c\}$, i.e. $ISS = \{not\ r, not\ \neg c\}$. So we are able to conclude c despite the inconsistency potentially caused by the other rules.

Note that according to *WFSX* this program is contradictory.

The most sceptical well–founded semantics, or WFS0, is next construable as the *grounded* part of the ideal sceptical semantics. Indeed, in the case of normal programs, the ideal sceptical semantics is determined as the greatest lower bound of all preferred extensions [Dung, 1991], well–founded semantics being the grounded part of this ideal sceptical semantics. This corroborates the intuitions of other related fields, where a distinction is made between restricted and ideal scepticism [Stein, 1989][8].

In this context, in order to define the well–founded sceptical semantics for programs with explicit negation, all we need is introduce the grounded part of ideal scepticism:

Definition 7.2.2 (WFS0) *Let P be an extended logic program whose ideal sceptical semantics is $P \cup H$. First define a transfinite sequence $\{K_\alpha\}$ of sets of hypotheses of P :*

$$\begin{array}{rcl} K_0 & = & \{\} \\ K_{\alpha+1} & = & K_\alpha \cup (H \cap MA(K_\alpha)) \end{array}$$

where

$$MA(K_\alpha) = Mand(K_\alpha) \cup Acc(K_\alpha).$$

The well–founded (sceptical) semantics of P, denoted WFS0, is defined as:

$$P \cup \bigcup_\alpha K_\alpha$$

[8] One other example of such restricted scepticism in logic programming is the "well–founded semantics wrt *Opt*" presented in chapter 8, which is even more sceptical then the aforementioned WFS0.

Hypotheses belonging to WFS0 belong perforce to ISS, because that is imposed at each step of the above iterative process by $MA(K_\alpha)$, and are also grounded in the sense that they are obtained by this bottom–up process starting from $\{\}$.

Example 7.10 Consider program P :

$$\begin{array}{lcl} a & \leftarrow & not\ a \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

Apart from literals $not\ \neg a$, and $not\ \neg b$ which are irrelevant to this example, admissible scenaria are:

$$P \cup \{\} \qquad P \cup \{not\ b\}^9$$

Thus $ISS = \{not\ b\}$.

In order to calculate the WFS0 let us build the sequence:

- By definition $K_0 = \{\}$.
- Since the program is normal there are no mandatories wrt $P \cup \{\}$.
  - $not\ b$ is not acceptable because $\{not\ a\}$ is evidence for b not defeated by $P \cup \{\}$, i.e. $P \cup \{\} \not\vdash a$;
  - Similarly, $not\ a$ is also not acceptable.

Thus $MA(K_0) = Mand(\{\}) \cup Acc(\{\}) = \{\}$, and

$$K_1 = \{\} \cup (\{not\ b\} \cap \{\}) = \{\} = K_0$$

So $WFS0 = P \cup \{\}$ because $not\ b$ is not grounded.

Theorem 7.2.1 *WFS0 is defined uniquely for every consistent program.*

Proof: Trivial since, as stated above, ISS is defined for every consistent programs and WFS0 is obtained uniquely from ISS. $\diamondsuit$

The next theorem states this definition of well–foundedness is a generalization of the one for non–extended (i.e. normal) programs.

Theorem 7.2.2 (Relation to the WFS of normal programs) *If P is a normal program then the WFS0 and the the well–founded semantics of [Gelder* et al.*, 1991] coincide.*

Proof: Clearly, if a program P has no explicit negation for every scenario $P \cup H$

$$Mand(H) = \{\}$$

Thus the definitions of evidence to the contrary, acceptability, and admissible scenario are equivalent to those for normal programs presented in [Dung, 1991]. So the ideal sceptical semantics corresponds to the intersection of preferred extensions and, as proven in [Dung, 1991], its grounded part coincides with the well–founded semantics of [Gelder *et al.*, 1991]. $\diamondsuit$

[9] Note that scenario $P \cup \{not\ a\}$ is inconsistent.

7.3 The semantics of complete scenaria

In this section we present a semantics less sceptical than WFS0 but failing to give semantics to all consistent programs. We call it *"complete scenaria semantics"* (CSS for short). Then we exhibit and prove some properties of CSS, in particular that it coincides with *WFSX*.

For normal programs every acceptable hypothesis can be accepted. In extended programs an acceptable hypotheses may fail to be accepted, in case a contradiction is verified.

Example 7.11 Consider the consistent program P :

$$\begin{array}{rcl} \neg a & & \\ a & \leftarrow & not\ b \end{array}$$

The hypothesis *not b* is acceptable wrt every scenario of P. However, by accepting *not b* the program becomes inconsistent. Thus *not b* can never be accepted. In a semantics like WFS0 such hypotheses are not accepted.

ISS and WFS0 model a reasoner who assumes the program correct and so, whenever confronted with an acceptable hypothesis leading to an inconsistency he cannot accept such a hypothesis; he prefers to assume the program correct rather than assume that an acceptable hypothesis must be accepted (cf. example 7.9 where both *not p* and *not q* are acceptable, but not accepted). We can also view this reasoner as one who has a more global notion of acceptability. For him, as usual, an hypothesis can only be acceptable if there is no evidence to the contrary, but if by accepting it (along with others) a contradiction arises, then that counts as evidence to the contrary.

It is easy to imagine a less sceptical reasoner who, confronted with an inconsistent scenario, prefers considering the program wrong rather than admitting that an acceptable hypothesis be not accepted. Such a reasoner is more confident in his acceptability criterium: an acceptable hypothesis is accepted once and for all; if an inconsistency arises then there is certainly a problem with the program, not with the acceptance of each acceptable hypothesis. This position is justified by the stance that acceptance be grounded on the absence of specific contrary evidence rather than on the absence of global non–specific evidence to the contrary. We come back to this issue in chapter 8, where we compare the more sceptical semantics with a revision process acting over the less sceptical one.

In order to define a semantics modeling the latter type of reasoner we begin by defining a subclass of the admissible scenaria, which directly imposes that acceptable hypotheses are indeed accepted.

Definition 7.3.1 (Complete scenario) *A scenario $P \cup H$ is complete iff is consistent, and*

$$H = Mand(H) \cup Acc(H)$$

i.e. $P \cup H$ is complete iff is consistent, and for each not L :

$$\begin{array}{lrcl} (i) & not\ L \in H & \Rightarrow & not\ L \in Acc(H) \vee not\ L \in Mand(H) \\ (ii) & not\ L \in Mand(H) & \Rightarrow & not\ L \in H \\ (iii) & not\ L \in Acc(H) & \Rightarrow & not\ L \in H \end{array}$$

where (i) and (ii) jointly express admissibility.

Example 7.12 The only complete scenario of program P :

$$\begin{array}{rcl} \neg b & & \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ c \\ a & \leftarrow & b, not\ \neg b \end{array}$$

is $P \cup \{not\ a, not\ \neg a, not\ b, not\ \neg c\}$. In fact:

- the mandatory hypotheses of that scenario are $\{not\ b\}$;
- $not\ \neg a$ is acceptable because there is no evidence for $\neg a$;
- $not\ \neg c$ is acceptable because there is no evidence for $\neg c$;
- $not\ a$ is acceptable because $not\ \neg b$ belongs to every evidence for a, and $\neg b$ is entailed by the scenario;
- $not\ c$ is not acceptable because $\{not\ c\}$ is evidence for c.

Since every acceptable or mandatory hypothesis is in the scenario, and every hypothesis in the scenario is either acceptable or mandatory, the scenario is complete.

Mark that if $not\ \neg b$ were not part of the second rule, as required by definition 2.1.1 of canonical program, then $not\ a$ would not be acceptable.

As expected, and in contradistinction to WFS0, complete scenaria may in general not exist, even when the program is consistent.

Example 7.13 Program P :

$$\begin{array}{rcl} \neg a & \leftarrow & not\ b \\ a & \leftarrow & not\ c \end{array}$$

has several admissible scenaria:

$$\begin{array}{lllllllll} P & \cup & \{\} & & P & \cup & \{not\ b\} & & P & \cup & \{not\ c\} \\ P & \cup & \{not\ a, not\ b\} & & P & \cup & \{not\ \neg a, not\ c\} & & & & \end{array}$$

None is complete. For example $P \cup \{not\ \neg a, not\ c\}$ is not complete because $not\ b$ is acceptable wrt that scenario.

Definition 7.3.2 (Contradictory program) *A program is contradictory iff it has no complete scenaria.*

Definition 7.3.3 (Complete scenaria semantics) *Let P be a noncontradictory program.*
The complete scenaria semantics of P is the set of all complete scenaria of P.
As usual, the meaning of P is determined by the intersection of all such scenaria.

The inexistence of semantics for some consistent programs might be seen as showing the inadequacy of CSS in certain cases, specially if compared to WFS0. As we will see in chapter 8, this is not the case since less sceptical semantics can be captured using CSS[10] and a revision process. The rationale of this view is:
"If an inconsistency arises then there is certainly a problem with the program, not with the acceptance of each acceptable hypothesis. If the problem is with the program then its revision is in order."

By using CSS one can rely on structural properties that, unlikely those of WFS0, make it amenable for devising bottom–up and top–down procedures, and also allow for more favourable computational complexity results (cf. chapter 10).

[10] In chapter 8 we use *WFSX* instead of CSS. However, as we prove afore, these semantics coincides.

7.4 Properties of complete scenaria

In this section we study some properties of this semantics, present a fixpoint operator for it, and show its relationship with *WFSX*.

Theorem 7.4.1 *Let $CS_P \neq \{\}$ be the set of all complete scenaria of noncontradictory program P. Then:*

1. *CS_P is a downward–complete semilattice, i.e. each nonempty subset of CS_P has a greatest lower bound.*
2. *There exists a least complete scenario.*
3. *In general, CS_P is not a complete partial order[11], i.e. maximal elements might not exist.*

For the sake of simplicity the proof of this theorem is in appendix. However we would like to present here an example showing that in general maximal complete scenario might not exist (viz. point 3 above):

Example 7.14 Consider the program:

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ \neg a & \leftarrow & not\ b \\ b & \leftarrow & not\ p(X) \\ p(X) & \leftarrow & not\ q(X) \\ q(X) & \leftarrow & not\ p(X) \end{array}$$

with Herbrand base $\mathcal{H} = \{0, 1, 2, 3, \ldots\}$.

For this program every set of the form

$$S_i = \{not\ q(k) \mid k \leq i\}$$

is a complete scenario, but there exists no complete scenario containing

$$\bigcup_i S_i.$$

Given that a least scenario always exists, we define:

Definition 7.4.1 (Well–founded complete scenario) *Let P be noncontradictory. The well–founded complete scenario $WF(P)$, is the least complete scenario of P.*

For this semantics we define an operator over scenaria such that every fixpoint of it is a complete scenario.

Definition 7.4.2 (V_P operator) *Given a program P and a set of hypotheses H we define:*

$$V_P(H) = H \cup Mand(H) \cup Acc(H)$$

just in case $P \cup V_P(H)$ is a consistent scenario; otherwise $V_P(H)$ is not defined.

The correctness of this operator is shown by the following (trivial) lemma.

Lemma 7.4.2 *$P \cup H$ is a complete scenario iff $H = V_P(H)$.*

[11] However, for normal programs CS_P is a complete partial order.

Another important result regarding the properties of the V_P operator is:

Lemma 7.4.3 *V_P is monotonic, by construction of its parts.*

From this lemma, and point 2 of theorem 7.4.1, it follows that:

Theorem 7.4.4 *If P is noncontradictory then the least fixpoint of V_P is the $WF(P)$.*

Theorem 7.4.5 (Iterative construction of the WF complete scenario) *In order to obtain a constructive bottom–up iterative definition of the WF scenario of a noncontradictory program P, we define the following transfinite sequence $\{H_\alpha\}$ of sets of hypotheses of P:*

$$\begin{aligned} H_0 &= \{\} \\ H_{\alpha+1} &= V_P(H_\alpha) \\ H_\delta &= \bigcup\{H_\alpha \mid \alpha < \delta\} \quad \textit{for a limit ordinal } \delta \end{aligned}$$

By lemma 7.4.3 and the Knaster–Tarski theorem [Tarski, 1955], there exists a smallest ordinal λ such that H_λ is a fixpoint of V_P. The WF complete scenario is $P \cup H_\lambda$.

This constructive definition obliges one to know *a priori* whether a program is contradictory. This prerequisite is not needed if we employ the following theorem.

Theorem 7.4.6 *A program P is contradictory iff in the sequence of the H_α there exists a λ such that $P \cup V_P(H_\lambda)$ is an inconsistent scenario.*

Thus, in order to compute the $WF(P)$ start building the above sequence. If, at some step i, H_i introduces a pair of complementary objective literals then end the iteration and P is contradictory. Otherwise iterate until the least fixpoint of V_P, which is the $WF(P)$.

Note the similarities between this process and the one described in section 6.7 for *WFSX*, where the iteration also provides the default literals *not* F (here caled hypotheses) true in the model, other literals T being determined by the former (there $T = \Gamma(\mathcal{H} - F)$, and here $T = \{L \mid P \cup not\ F \vdash L\}$).

7.4.1 Complete scenaria and *WFSX*

Next we present the relationship between the complete scenaria semantics CSS for extended logic programs and *WFSX*, showing they are the same. The significance of this result is underscored in the introduction to this chapter. Proofs of lemmas can be found in appendix C.

Lemma 7.4.7 (*PSM*s correspond to complete scenaria) *Let*

$$S = T \cup not\ F$$

be a PSM of a program P, where T and F are disjoint sets of objective literals. Then:

$$P \cup not\ F$$

is a complete scenario.

Lemma 7.4.8 (Complete scenaria correspond to *PSM*s) *If*

$$P \cup H$$

is a complete scenario then:

$$\{L \mid P \cup H \vdash L\} \cup H$$

is a PSM of P.

Theorem 7.4.9 (Equivalence) *The complete scenaria semantics CSS is equivalent to* WFSX.

7.5 More credulous semantics

Along the same lines of complete scenaria semantics, we can continue restricting the set of admissible scenaria, thus defining more credulous semantics.

The most immediate semantics more credulous than CSS (or *WFSX*) is the one obtained by considering only maximal (wrt $\subseteq$) complete scenaria. We call this semantics "preferred extensions" following the tradition for normal programs [Dung, 1991].

Definition 7.5.1 (Preferred extensions semantics) *The preferred extensions semantics of an extended program P is the set of its maximal complete scenaria.*

Example 7.14 shows that maximal elements might not exist for a collection of complete scenaria, hence preferred extensions are defined for less programs than *WFSX*. Another straightforward result is that this semantics is in general more credulous than *WFSX*.

Example 7.15 Consider the program:

$$\begin{array}{rcl} a & \leftarrow & not\ p, not\ \neg p \\ p & \leftarrow & not\ \neg p \\ \neg p & \leftarrow & not\ p \end{array}$$

Complete scenaria are (where the last two are preferred):

$$\begin{array}{l} P \cup \{not\ \neg a\} \\ P \cup \{not\ \neg a, not\quad p, not\ a\} \\ P \cup \{not\ \neg a, not\ \neg p, not\ a\} \end{array}$$

Thus *not a* is a consequence of the preferred extensions semantics but not of complete scenaria semantics.

A reasoner can even be more credulous by considering only preferred extensions that are two valued (or total), i.e. extensions such that whenever L is not a consequence of them *not* L is assumed in them.

Definition 7.5.2 (Total scenario) *A scenario $P \cup H$ is total iff for every objective literal L :*

$$P \cup H \vdash L \quad \equiv \quad not\ L \notin H$$

Definition 7.5.3 (Total scenaria semantics) *The total scenaria semantics of an extended program P is the set of its total complete scenaria.*

Given the results of [Dung, 1991], where stable models are total complete scenaria in normal logic programs, it follows easily:

Theorem 7.5.1 (Answer–sets) *The total scenaria semantics coincides with the answer–sets semantics of [Gelfond and Lifschitz, 1990].*

Clearly answer–sets semantics is defined for less programs than the previous semantics, since such total scenaria may in general not exist. The typical program for which answer–sets semantics is not defined but *WFSX* is defined is $P = \{a \leftarrow not\ a\}$, where assuming *not a* leads to an inconsistency between a and *not a*, and *not a* cannot be left unassumed because a is not a consequence. This program has only one complete scenario, $\{not\ \neg a\}$, and it is not total.

Explicit negation introduces other cases of inexistence of answer–sets appear.

Example 7.16 Let P be:

$$\begin{array}{rcl} p & \leftarrow & not\ \neg p \\ \neg p & \leftarrow & not\ p \\ b & \leftarrow & not\ \neg p \\ a & \leftarrow & not\ p \\ \neg a & & \\ \neg b & & \end{array}$$

The only complete scenario is $P \cup \{not\ a, not\ b\}$, which is not total. Thus no answer–sets exist.

Here the inexistence of answer–sets is due to inconsistency between an objective literal and its explicit negation:

- assuming *not* p leads to an inconsistency between a and $\neg a$;
- the assumption *not* p can be dropped only if p is a consequence. In order to make p a consequence *not* $\neg p$ must be assumed, and then an inconsistency between b and $\neg b$ appears.

Example 7.14 shows additional issues regarding the existence of answer–sets. In particular that example shows that the computation of an answer–set cannot in general be made by finite approximations.

7.5.1 Comparisons among the semantics

From the definition 7.2.2 of WFS0 and the iterative construction of the WF complete scenario of CSS (theorem 7.4.5) it follows almost directly that:

Theorem 7.5.2 (WFS0 is more sceptical than *WFSX*) *For any noncontradictory program P*

$$WFS0(P) \subseteq \mathrm{WFSX}(P).$$

Example 7.17 Consider program P :

$$\begin{array}{rcl} p & \leftarrow & not\ q \\ \neg p & \leftarrow & a \\ \neg p & \leftarrow & b \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

whose *WFSX* is $\{not\ q\}$ (apart from irrelevant literals such as *not* $\neg a$).

Since $P \cup \{not\ q, not\ \neg p\}$, $P \cup \{not\ a, not\ p\}$, and $P \cup \{not\ a, not\ p\}$ are all admissible scenaria (though not them all), and neither *not* a nor *not* b can be added to the first scenario, and also *not* q cannot be added neither to the second nor to the third scenario above, then $ISS = \{\}$. Thus $WFS0 = \{\}$.

Interesting questions are: *When do all these semantics coincide? Can we state sufficient conditions guaranteeing such an equivalence?*

In order to answer the second question we introduce the notion of semantically normal (s–normal for short) programs; i.e. those whose admissible scenaria can all be completed.

Definition 7.5.4 (S–normal program) *An extended program is s–normal iff for each admissible scenario $P \cup H$:*

$$P \cup H \cup Acc(H)$$

is consistent.

Lemma 7.5.3 *Let P be a s–normal program, $P \cup H$ be an admissible scenario, and let not A, not B be acceptable wrt $P \cup H$. Then:*

1. *$P \cup H \cup \{not\ A\}$ is admissible and*
2. *not B is acceptable wrt $P \cup H \cup \{not\ A\}$.*

Proof: Trivial, given the definition of s–normal program. $\diamondsuit$

From this lemma it follows immediately that the set of all admissible scenarios (wrt set inclusion) forms a complete partial order for s–normal programs. Hence each admissible scenario can be extended into a complete scenario. Thus, for s–normal programs, ISS is contained in a complete scenario.

Moreover, it is easy to see that for each admissible scenario $P \cup H$, $P \cup H \cup CSS(P)$ is again admissible. Therefore:

Theorem 7.5.4 *Let P be a s–normal program. Then:*

- *The set of complete scenaria of P forms a complete semilattice.*
- *ISS coincides with the intersection of preferred extensions.*
- $WFS0(P) = CSS(P) \subseteq ISS(P)$.

To define larger classes of programs also guaranteeing these comparability results is beyond the scope of this work. Of special interest, and subject of future investigation by the author, is to determine syntatic conditions over programs (e.g. a generalization of the notion of stratified normal programs [Apt *et al.*, 1988]) guaranteeing the equivalence between answer-sets and *WFSX*, in the vein of the work in [Dung, 1992b] regarding well founded and stable models semantics of normal programs.

However, for normal logic programs, since acceptable hypotheses can never lead to an inconsistency, both WFS0 and *WFSX* coincide.

Theorem 7.5.5 (Relation to the WFS of normal programs) *If P is a normal (non–extended) program then* WFSX*, WFS0 and the well–founded semantics of [Gelder* et al.*, 1991] coincide.*

Example 7.10 shows this equivalence cannot be extended to ISS. There, *WFSX* coincides with WFS0 and with WFS and is $\{\}$. ISS is $\{not\ b\}$.

Chapter 8

Dealing with contradiction

As we've seen before, *WFSX* is not defined for every program, i.e. some programs are contradictory and are given no meaning[1]. While for some programs this seems reasonable (e.g. example 4.5 in page 33), for others this can be too strong.

Example 8.1 Consider the statements:

- *Birds, not shown to be abnormal, fly.*
- *Tweety is a bird and does not fly.*
- *Socrates is a man.*

naturally expressed by the program:

$$fly(X) \leftarrow bird(X), not\ abnormal(X).$$

$$bird(tweety)$$
$$\neg fly(tweety).$$

$$man(socrates).$$

WFSX assigns no semantics to this program. However, intuitively, we should at least be able to say that *Socrates* is a *man* and *tweety* is a *bird*. It would also be reasonable to conclude that *tweety* doesn't fly, because the rule stating that it doesn't fly, since it is a fact, makes a stronger statement than the one concluding it $flies$. The latter relies on accepting an assumption of non–abnormality, enforced by the closed world assumption treatment of the negation as failure, and involving the abnormality predicate. Indeed, whenever an assumption supports a contradiction it seems logical to be able to take the assumption back in order to prevent it – *"Reductio ad absurdum"*, or *"reasoning by contradiction"*.

In chapter 7 we present semantics more sceptical than *WFSX*, that avoid contradiction in many cases where the latter gives no meaning to a program. For example ISS assigns to the above program the meaning (with the obvious abbreviations for constants):

$$\{man(s), \neg fly(t), bird(t), not\ fly(t)\}$$

which exactly corresponds to the intuition above.

[1] Other researchers have defined paraconsistent semantics for even contradictory programs e.g. [Costa, 1974, Blair and Subrahmanian, 1987, Kifer and Lozinskii, 1989, Sakama, 1992, Wagner, 1993]. This is not our concern. On the contrary, we wish to remove contradiction whenever it rests on withdrawable assumptions.

Furthermore, there is motivation to consider even more sceptical semantics, where some of the acceptable assumptions or hypotheses might not in fact be accepted.

For instance, the acceptance of a hypothesis may be conditional upon the equal acceptance of another. This is typical of hypothesizing faults in a device, whenever causally deeper faults are to be preferred over hypothesized faults that are simply a consequence of the former: the latter cannot be hypothesized without the first. Moreover, problem specific and user defined preference criteria affecting acceptance of hypotheses may also come to bear. Another case in point is logic program debugging, where one wants to hypothesize about the primitive cause of a bug, and not about the bugginess of some clause, if there is the possibility that that clause relies in fact on a still buggy predicate [Pereira *et al.*, 1993d, Pereira *et al.*, 1993e, Pereira *et al.*, 1993c]. In general, the clauses of a logic program may be seen as providing a causal directionality of inference, similar to physical causality directionality, so that a distinction can sometimes be drawn about the primacy of one hypothesis over another, cf. [Konolige, 1992, Brewka and Konolige, 1993].

Example 8.2 Consider this program, describing bycicle behaviour:

$$\begin{aligned} \neg wobbly\_wheel &\leftarrow not\ flat\_tyre, not\ broken\_spokes \\ flat\_tyre &\leftarrow leaky\_valve \\ flat\_tyre &\leftarrow punctured\_tube \\ \neg no\_light &\leftarrow not\ faulty\_dynamo \end{aligned}$$

plus the factual observation:

$$wobbly\_wheel$$

The ISS assigns to it the meaning:

$$\{wobbly\_wheel, not\ faulty\_dynamo, \neg no\_light, not\ no\_light, \\ not\ leaky\_valve, not\ punctured\_tube\}$$

neither accepting the hypothesis *not flat_tyre* nor *not broken_spokes* because acceptence of any of them, if the other were accepted too, would lead to a contradiction. Being sceptical ISS accepts neither. However, one would like the semantics in this case to delve deeper into the bycicle model and, again being sceptical, accept neither *not leaky_valve* nor *not punctured_tube* as well.

In order to respond to such epistemological requirements as above, we begin by introducing into the complete scenario semantics the more flexible notion of optative acceptance of hypotheses. Optative hypotheses are those that might or might not be accepted if acceptable at all. On the other hand, non–optative hypotheses must be accepted if acceptable.

First we make no restriction on what the optatives are, and consider that they are given by the user along with the program. Then we proceed to consider the issue of infering optative hypotheses from the program, given some specific criteria. In particular we show how to infer optatives when the criteria is to consider as such those hypotheses that do not depend on any other[2].

As claimed before, these very sceptical semantics model rational reasoners who assume the program absolutelly correct and so, whenever confronted with an acceptable hypothesis leading to an inconsistency cannot accept such a hypothesis; i.e. they prefer to assume the program correct rather than assume that an acceptable hypothesis must perforce be accepted.

WFSX models less sceptical reasoners who, confronted with an inconsistent scenario, prefer considering the program wrong rather than admitting that an acceptable hypothesis be not

[2]Considered above as the preferred criterium for the case of fault finding, and debugging.

accepted. Such a reasoner is more confident in his acceptability criterium: an acceptable hypothesis is accepted once and for all; if an inconsistency arises then there is certainly a problem with the program, not with the individual acceptance of each acceptable hypothesis. If the problem is with the program its revision is in order.

This view position can be justified if we think of a program as something dynamic, i.e. evolving in time. In this position each program results from the assimilation of knowledge into a previous one. If an inconsistency arises from the knowledge assimilation then a revision process should be considered so as to restore consistency.

In [Kowalski, 1990], Kowalski presents a detailed exposition of the intended behaviour of this knowledge assimilation processes in various cases. There he claims the notion of integrity constraints is needed in logic programming both for knowledge processing, representation, and assimilation. The problem of inconsistency arises from nonsatisfaction of the integrity constraints. If some new knowledge can be shown incompatible with the existing theory and integrity constrains, a revision process is needed to restore satisfaction of those constraints.

In extended logic programming we can view the requirement of noncontradiction as integrity constraint satisfaction, where constraints are of the form $\leftarrow L, \neg L$. But then there is no reason why we should not allow a more general form of integrity contraints. In this chapter we extend logic programs with integrity constraints in the form of denials.

Example 8.3 Suppose we have some program describing political affiliation and don't want to say that non democrats are republicans and vice–versa. Thus $\neg republican(X)$ should not correspond to $democrat(X)$ and $\neg democrat(X)$ should not correspond to $republican(X)$. However, no one must be known both as a republican and a democrat. This knowledge can be easily represented by the integrity constraint:

$$\leftarrow democrat(X),\ republican(X)$$

Let's go back now to example 8.1. We can also view that program as the result of knowledge assimilation into a previous knowledge base expressed by a program. For example the program can be thought of as the adding to the previous knowledge the fact that tweety does not fly. According to *WFSX* the resulting program is inconsistent. One way of restoring consistency to the program would be to add a rule stating that $ab(tweety)$ cannot be false, viz. it would lead directly to a contradiction:

$$ab(tweety) \leftarrow not\ ab(tweety)$$

The resulting program is now noncontradictory and its $WFSX$ is:

$$\{man(s), \neg fly(t), bird(t), not\ fly(t)\}$$

which corresponds to the intuition.

In this chapter we begin by presenting a sceptical semantics for extended logic programs plus integrity contraints in the form of denials, based on the notion of optative hypotheses, which avoids contradiction. We also define a program revision method for removing contradiction from contradictory programs under *WFSX*. Then, we show the equivalence between the (contradiction avoidance) semantics and the *WFSX* of the revised program obtained by the contradiction removal method. Finally, we show examples of application to diagnosis and to the debugging of pure Prolog programs.

Parts of this chapter appear in [Alferes and Pereira, 1993a], [Pereira and Alferes, 1993a] and in [Pereira and Alferes, 1993b].

8.1 Logic programming with denials

As argued by Reiter in [Reiter, 1990], the basic idea of integrity constraints is that only some program (or database) states are considered acceptable, and those constraints are meant to enforce these acceptable states.

Integrity constraints can be of two types:

Static The enforcement of these constraints depends only on the current state of the program, independently of any prior state. The democrat/republican constraint above is one such example.

Dynamic These depend on two or more program states. In [Reiter, 1990], Reiter gives as example the knowledge that employee salaries can never decrease.

It is not a purpose of this work to deal with the evolution of a program in time. Thus dynamic integrity contraints are not addressed. Since we only want to deal with the problem of inconsistency, it is enough that the only static integrity constraints considered be in the form of denials. For a study of different forms of static constraints and their satisfaction see [Reiter, 1990].

Next we formally define the language of extended logic programs plus denials, and the notion of integrity contraint satisfaction adopted in this chapter.

A program with integrity rules (or constraints) is a set of rules as defined in section 2.1, plus a set of denials, or integrity rules, of the form:

$$\bot \leftarrow A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m$$

where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are objective literals, and $n + m > 0$. The symbol $\bot$ stands for falsity.

A program P with a semantics SEM satisfies the integrity constrains iff:

$$P \not\models_{SEM} \bot$$

8.2 Contradiction avoidance

In this section we present a semantics more sceptical than ISS, based on the notion of scenaria described in section 7. Thus the attending notions of program transformation (in order to obtain only Horn programs),of consequence given a scenario, etc., all apply here.

To deal with denials we extend the notion of consistent scenario.

Definition 8.2.1 (Consistent scenario wrt ICs) *A scenario $P \cup H$ of a program with integrity constraints IC is consistent iff:*

- *for all objective literals L such that:*

  $$P \cup H \cup Mand(H) \vdash L,$$

  neither

  $$not\ L \in H \cup Mand(H) \text{ nor } P \cup H \cup Mand(H) \vdash \neg L,$$

  and

- $P \cup H \cup Mand(H) \cup IC \not\vdash \bot$[3].

[3]ICs are treated like any other rule for deriving $\bot$, hence the designation of *"integrity rule"*.

If one implicitly adds to a program P constraints of the forms:

$$\bot \leftarrow L, not\ L$$

for every objective literal L of P, then the first condition above is obviously subsumed by the second one, and thus can be withdrawn.

Proposition 8.2.1 *A scenario $P \cup H$ of a program with integrity constrains IC is consistent iff:*

$$P \cup H \cup Mand(H) \cup NIC \not\vdash \bot$$

where:

$$NIC = IC \cup \{\bot \leftarrow L, not\ L;\ \bot \leftarrow L, \neg L \mid L \in lang(P)\}$$

Like for extended logic programs before, an extended logic program with denials may have no consistent scenaria.

Example 8.4 Program P :

$$\begin{array}{rcl} \neg democrat(husband(mary)) & \leftarrow & \\ republican(mary) & \leftarrow & \\ democrat(X) & \leftarrow & \neg democrat(husband(X)) \\ & & \\ \bot & \leftarrow & democrat(X), republican(X) \end{array}$$

has no consistent scenario.

Definition 8.2.2 (Consistent program with ICs) *An extended logic program P with integrity constraints IC is consistent iff it has some consistent scenario.*

N.B. From now on, unless otherwise stated, we restrict programs to consistent ones only.

In *WFSX* every acceptable hypothesis must be accepted. Consequently some programs might have no meaning. In ISS some acceptable hypotheses are not accepted in order to avoid inconsistency. However, as shown in example 8.2, ISS allows no the control over which acceptable hypotheses are not accepted. Conceivably, any acceptable hypothesis may or may not actually be accepted, in some discretionary way.

It is clear from example 8.2 that we wish to express that only the hypotheses

$$not\ broken\_spokes,\ not\ leaky\_valve,\ not\ faulty\_dynamo \text{ and } not\ punctured\_tube$$

may be optative, i.e. to be possibly accepted or not, if at all acceptable. The acceptance of hypotheses like $not\ flat\_tyre$ is to be determined by the acceptance of other hypotheses, and so we wish them accepted once acceptable.

Thus we should distinguish between *optative hypotheses* (or optatives) and non–optative ones. That distinction made, we can conceive of scenaria that might not be complete wrt optatives, but are still complete wrt non–optatives, i.e. scenaria which contain all acceptable hypotheses except for possibly optative ones.

Definition 8.2.3 (Optative hypotheses) *The set of optative hypotheses Opt is any subset of not $\mathcal{H}$.*

In general, when not accepting some optative hypothesis $not\ L$, i.e. when not assuming the falsity of L, then some otherwise acceptable hypotheses become unacceptable. The sense desired is that program models where the optative is true are not ruled out.

Example 8.5 Let P :

$$\begin{array}{lcl} p & \leftarrow & not\ a \\ a & \leftarrow & b \\ \\ \bot & \leftarrow & p \end{array}$$

where *not* b is the only optative, i.e. $Opt = \{not\ b\}$.

In our notion of optative, if *not* b is not accepted then *not* a is unacceptable, i.e. if optative b is not assumed false, the possibility of being true must be considered and so a cannot be assumed false; $P \cup \{b\} \vdash a$ counts as evidence against *not* a.

Definition 8.2.4 (Acceptable hypothesis wrt *Opt***)** *A hypothesis not* L *is acceptable wrt scenario* $P \cup H$ *and set of optatives Opt iff*

$$not\ L\ is\ acceptable^4\ both\ wrt\ P \cup H\ and\ P \cup H \cup F$$

where F *is the set of facts*

$$not\ ((Opt \cap Acc(H)) - H)$$

i.e. F *is the set of complements of acceptable Opts wrt* H *which are not in* H *(that is which were not accepted).*

$Acc_{Opt}(H)$ *denotes the set of acceptable hypotheses wrt* $P \cup H$ *and Opt.*

Example 8.6 In example 8.5 $Acc_{Opt}(\{not\ p\}) = \{\}$.

not b is not acceptable because, even though acceptable wrt $P \cup \{not\ p\}$, it is not acceptable wrt $P \cup \{not\ p\} \cup \{b\}$[5]. The same happens with *not* a.

With this new more general notion of acceptability, we can define scenaria that are partially complete, in the sense that they are complete wrt non–optatives, but might not be complete wrt optatives (condition (iii) below).

Definition 8.2.5 (Complete scenario wrt *Opt***)** *A scenario* $P \cup H$ *is a complete scenario wrt a set of optatives Opt iff it is consistent, and for each not* L :

$$\begin{array}{ll} (i) & not\ L \in H \Rightarrow not\ L \in Acc_{Opt}(H) \vee not\ L \in Mand(H) \\ (ii) & not\ L \in Mand(H) \Rightarrow not\ L \in H \\ (iii) & not\ L \in Acc_{Opt}(H)\ and\ not\ L \notin Opt \Rightarrow not\ L \in H \end{array}$$

Remark 8.2.1 *By making* $Opt = \{\}$ *the previous definitions of acceptability wrt Opt and of complete scenaria wrt Opt correspond exactly to those of acceptability and complete scenaria in section 7.*

By making $Opt = not\ \mathcal{H}$ *the definitions of acceptability wrt Opt and of complete scenaria wrt Opt correspond exactly to those of acceptability and admissible scenaria in section 7.*

Note that in complete scenario $S = P \cup H$ wrt *Opt* a hypothesis in *Opt* which is acceptable wrt $P \cup H$ but leads to an inconsistent scenario, will not be accepted in S to preserve consistency. This amounts to contradiction avoidance.

Example 8.7 Recall the wobbly wheel example 8.2. If *Opt* were $\{\}$ there would be no complete scenaria. If (with the obvious abbreviations):

$$Opt = \{not\ bs, not\ lv, not\ pt, not\ fd\}$$

[4]Acceptable cf. definition 7.1.7.

[5]Note that here $not\ ((Opt \cap Acc(H)) - H) = not\ (\{not\ b\} - \{not\ p\}) = \{b\}$.

complete scenaria wrt *Opt* are :

$$\begin{array}{cc} \{not\ \neg ww\} & \{not\ \neg ww, not\ fd, not\ bs\} \\ \{not\ \neg ww, not\ fd\} & \{not\ \neg ww, not\ lv, not\ pt, not\ ft\} \\ \{not\ \neg ww, not\ bs\} & \{not\ \neg ww, not\ fd, not\ lv\} \\ \{not\ \neg ww, not\ lv\} & \{not\ \neg ww, not\ lv, not\ pt, not\ ft, not\ fd\} \\ \{not\ \neg ww, not\ pt\} & \ldots \end{array}$$

Intuitively, it is clear that some of these scenaria are over–sceptical, in the sense that they fail to accept more optatives than need be to avoid contradiction. For example in the first scenario in order to avoid contradiction none of the optatives where accepted. This occurs because no condition of maximal acceptance of optatives has been enforced.

In order to impose this condition we begin by identifying, for each complete scenario wrt *Opt*, those optatives that though acceptable were not accepted.

Definition 8.2.6 (Avoidance set) *Let $P \cup H$ be a complete scenario wrt Opt. The avoidance set of $P \cup H$ is (the subset of Opt):*

$$(Opt \cap Acc(H)) - H$$

Example 8.8 The avoidance set of the first scenario in example 8.7 is:

$$\{not\ lv, not\ pt, not\ fd\}$$

and of the second one is:

$$\{not\ lv, not\ pt\}$$

In keeping with the vocation of scepticism of *WFSX*, we are specially interested in those scenaria which, for some given avoidance set, are minimal.

Definition 8.2.7 (Base scenario wrt *Opt*) *A complete scenario $P \cup H$ wrt Opt, is a base scenario if there exists no scenario $P \cup H'$ with the same avoidance, set such that $H' \subset H$.*

Example 8.9 Consider the program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ c & \leftarrow & not\ d \\ \\ \bot & \leftarrow & c \end{array}$$

with $Opt = \{not\ d\}$.

Complete scenaria wrt *Opt* are:

$$\{\} \qquad \{a, not\ b\} \qquad \{b, not\ a\}$$

For all the avoidance set is $\{not\ d\}$. The corresponding base scenario wrt *Opt* is the first.

Proposition 8.2.2 *The set of all base scenaria wrt Opt under set inclusion forms a lower semi–lattice.*

Proof: Let $P \cup H_1$ and $P \cup H_2$ be two base scenaria with avoidance sets S_1 and S_2 respectively. We prove that there is a single maximal scenario $P \cup H$ such that $H \subseteq H_1$ and $H \subseteq H_2$.

Such a scenario must have an avoidance set $S \supseteq S_1 \cup S_2$. From the definition of complete scenario wrt *Opt* there exists one scenario such that its avoidance set $S = S_1 \cup S_2$. It is clear from lemma 8.4.1 below, that there is a least scenario with S as avoidance set. $\diamondsuit$

Consider now those scenaria comprising as many optatives as possible, i.e. have minimal avoidance sets:

Definition 8.2.8 (Quasi–complete scenario wrt *Opt*) *A base scenario $P \cup H$ wrt Opt, with avoidance set S, is quasi–complete if there is no base scenario $P \cup H'$ wrt Opt with avoidance set S', such that $S' \subset S$.*

Example 8.10 In example 8.7 the quasi–complete scenaria wrt *Opt* are:

$$\{not\ \neg ww, not\ fd, not\ bs, not\ lv\}$$
$$\{not\ \neg ww, not\ fd, not\ bs, not\ pt\}$$
$$\{not\ \neg ww, not\ fd, not\ lv, not\ pt, not\ ft\}$$

These correspond to minimal faults compatible with the wobbly wheel observation, i.e. the ways of avoiding contradiction (inevitable if *Opt* were {}) by minimally not accepting acceptable optatives. In the first *not pt* was not accepted, in the second *not lv*, and in the third *not bs*.

As the consequences of all these quasi–complete scenaria are pairwise incompatible[6] the well-founded model, being sceptical, is their meet in the semi–lattice of proposition 8.2.2, so that its avoidance set is the union of their avoidance sets.

Definition 8.2.9 (Well–founded semantics wrt *Opt*) *The well–founded model of an extended logic program P with ICs is the meet of all quasi–complete scenaria wrt Opt in the semi–lattice of all base scenaria.*

For short we use WFS_{Opt} to denote the well–founded model wrt Opt.

Example 8.11 In example 8.7 WFS_{Opt} is:

$$P \cup \{not\ \neg ww, not\ fd\}$$

Thus one can conclude:

$$\{ww, \neg nl, not\ \neg ww, not\ fd\}$$

i.e. no other hypothesis can be assumed for certain; everything is sceptically assumed faulty except for fd. This differs from the result of ISS, shown in example 8.2.

Example 8.12 Consider the statements:

- *Let's go hiking if it is not known to rain.*
- *Let's go swimming if it is not known to rain.*
- *Let's go swimming if the water is not known to be cold.*
- *We cannot go both swimming and hiking.*

They render the set of rules P:

$$\begin{aligned} hiking &\leftarrow not\ rain \\ swimming &\leftarrow not\ rain \\ swimming &\leftarrow not\ cold\_water \\ \\ \bot &\leftarrow hiking, swimming \end{aligned}$$

and let $Opt = \{not\ rain, not\ cold\_water\}$.

Complete scenaria wrt *Opt* are:

$$P \cup \{\} \qquad P \cup \{not\ cold\_water\}$$

where the latter is the well founded wrt *Opt*. It entails that *swimming* is true. Note that *not rain* is not assumed because it is optative to do so, and by assuming it contradiction would be unavoidable.

[6] In the sense that neither contains any other.

To obtain less sceptical complete scenaria wrt Opt, and in the spirit of the above described partial stable models, we introduce:

Definition 8.2.10 (Partial scenario wrt *Opt*) *Let P be an extended logic program with ICs, and let the well–founded semantics of P wrt Opt be $P \cup H$.*

$P \cup K$ is a partial scenario of P wrt Opt iff it is a base scenario wrt Opt and $H \subseteq K$.

Example 8.13 The partial scenaria of P wrt *Opt* in example 8.7 are the union of P with each of:

$$\begin{array}{ll} \{not\ \neg ww, not\ fd\} & \{not\ \neg ww, not\ fd, not\ bs, not\ lv\} \\ \{not\ \neg ww, not\ fd, not\ bs\} & \{not\ \neg ww, not\ fd, not\ bs, not\ pt\} \\ \{not\ \neg ww, not\ fd, not\ lv\} & \{not\ \neg ww, not\ fd, not\ lv, not\ pt, not\ ft\} \\ \{not\ \neg ww, not\ fd, not\ pt\} & \end{array}$$

The first is the WFS_{Opt} (cf. example 8.11), which corresponds to the most sceptical view whereby all possibly relevant faults are assumed. The other partial scenaria represent, in contrast, all other alternative hypothetical presences and absences of faults still compatible with the wobbly wheel observation.

If a program is noncontradictory (i.e. its *WFSX* exists) then no matter which are the optatives, the well-founded semantics wrt *Opt* is always equal to the least complete scenario (and so, ipso facto, equivalent to the *WFSX*).

Theorem 8.2.1 (Relation to *WFSX*) *If* WFSX *is defined for a program P with empty set of ICs then, for whatever Opt, WFS_{Opt} is the least complete scenario of P.*

Proof: If *WFSX* is defined for P then there exists at least one complete scenario of P. Thus there exists at least one complete scenario wrt *Opt*, $P \cup H$, such that its avoidance set is empty. So the only quasi–complete scenario, and WFS_{Opt}, is the base scenario with empty avoidance set.

By definition, the set of complete scenaria wrt *Opt* with empty avoidance set coincides with the set of complete scenaria, and thus the least complete scenario coincides with the base scenario wrt *Opt*. $\diamondsuit$

Since, cf. theorem 4.3.6, for programs without explicit negation *WFSX* is equivalent to the well–founded semantics of [Gelder *et al.*, 1991] (WFS):

Theorem 8.2.2 *Let P be a (non–extended) normal program. Then, for whatever Opt, the well–founded semantics wrt Opt is equivalent to its WFS.*

8.2.1 Primacy in optative reasoning

Up to now no restriction whatsoever was enforced regarding the optatives of programs. It is possible for optatives to be identified by the user along with the program, or for the user to rely on criteria for specifying the optatives, and expect the system to infer them from the program.

Next we identify a special class of optatives, governed by an important criterium [Konolige, 1992, Brewka and Konolige, 1993]:

Exactly the hypotheses not depending on any other are optative.

Example 8.14 Let P :

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ d \end{array}$$

Clearly *not a* depends on *not b*, *not b* on *not c* and *not c* on *not d*. *not d* alone does not depend on any other hypothesis, thus according to this criterium, it should be the only optative.

In diagnosis this criterium means hypothesizing as abnormal first the causally deeper faults.

It is known that in taxonomies with exceptions, this is not the desired preference criterium. To give priority to the most specific default information only a hypothesis on which no other depends should be optatives. This way the relinquishing of default hypotheses to avoid contradiction begins with less specific ones.

The subject of defining preference criteria to automatically determine optative hypotheses is complex. It is closely related to that of preference among defaults [Geerts and Vermeir, 1993].

The study of how to infer optatives for criteria different from the one above, is left as an open problem.

Clearly, every hypothesis which is not acceptable in $P \cup \{\}$ depends on the acceptance of some other hypothesis. In other words, if a hypothesis *not* L is acceptable in a scenario $P \cup H$, but is not acceptable in $P \cup \{\}$, this means that in order to make *not* L acceptable some other hypotheses $S \subseteq H$ have to be accepted first. Thus *not* L depends on the hypotheses of S, and the latter are more primal than *not* L. As a first approximation, let me define the set of prime optative hypotheses as $Acc(\{\})$.

Example 8.15 In program P of example 8.14 $Acc(\{\}) = \{not\ d\}$. So the only prime optative hypothesis is *not* d. Hypothesis *not* b is not prime optative because it is only acceptable once *not* d is accepted, otherwise *not* c constitutes evidence to the contrary.

In general, not all hypotheses in $Acc(\{\})$ though are independant of one another. Hence we must refine our first approximation to prime optatives.

Example 8.16 Consider P :

$$\begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & c \\ p & \leftarrow & not\ a \\ \\ \bot & \leftarrow & p \end{array}$$

$$Acc(\{\}) = \{not\ a, not\ b, not\ c\}$$

and the WFS wrt $Acc(\{\})$ is $P \cup \{not\ b, not\ c\}$.

However, it is clear from the program that only *not* c should be prime optative, since the acceptance of *not* b depends on the absence of conclusion c in P, but not vice–versa, and likewise regarding the acceptance of *not* a.

Any definition of a semantics based on the notions of scenaria and evidence alone cannot distinguish the optative primacy of *not* c, because it is insensitive to the groundedness of literals, viz. there being no rules for c, and thus its non–dependance on other hypotheses.

An assymmetry must be introduced, based on a separate new notion, to capture the causal directionality of inference implicit in logic program rules, as mentioned in the introduction to this chapter:

Definition 8.2.11 (Sensitive hypotheses) *A hypothesis not* $A \in Acc(\{\})$ *is sensitive to a separate set of hypotheses not* S *in program* P *iff*

$$not\ A \notin Acc(P \cup S)$$

Note that S *is a set of facts.*

Definition 8.2.12 (Prime optatives) *A hypothesis not $A \in Acc(\{\})$ is prime optative iff for all not $S \subseteq Acc(\{\})$:*

if not A is sensitive to not S then some element of not S is sensitive to not A.

The set of all prime optatives is denoted by $\mathcal{POpt}$.

As shorthand, we refer to the well–founded semantics wrt the set of prime optatives as the *prime optative semantics*, or $\mathcal{POS}$.

Example 8.17 In example 8.16 the only prime optative hypothesis is *not c*. For example, *not a* is not prime optative since *not a* is sensitive to *not b* and *not b* is not sensitive to *not a*.

Example 8.18 In the wobbly wheel example:

$$\mathcal{POpt} = \{not\ bs, not\ pt, not\ lv, not\ fd\}$$

For this example $Acc(\{\}) = \mathcal{POpt} \cup \{not\ ft\}$.
However *not ft* is not prime optative since it is sensitive to both *not lv* and *not pt*.

Example 8.19 Consider program P :

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} p & \leftarrow & not\ a & a & \leftarrow & b & c & \leftarrow & not\ d \\ \neg p & \leftarrow & & b & \leftarrow & a, not\ c & & & \end{array}$$

where:

$$Acc(\{\}) = \{not\ a, not\ b, not\ d\}$$

All of these are prime optatives:

- *not d* is prime optative because it is insensitive to other hypotheses;
- *not b* is prime optative because it is only sensitive to *not a*, and *not a* is sensitive to *not b*;
- similarly for *not a*.

By insisting on only allowing prime optatives to be possibly accepted, even if acceptable, one may fail to give meaning to some consistent programs, as there are less options for avoiding inconsistency.

Example 8.20 Consider program P :

$$\begin{array}{rcl} c & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ \neg a & & \\ \bot & \leftarrow & not\ c \end{array}$$

In this case $\mathcal{POpt} = Acc(\{\}) = \{not\ a\}$, and no complete scenario wrt $\mathcal{POpt}$ exists. Thus neither ISS wrt $\mathcal{POpt}$ nor $\mathcal{POS}$ are defined.

Note that by making $Opt = \{not\ c\}$, $P \cup \{not\ a\}$ is now complete wrt Opt. In fact this scenario correspont to the $WFM_{\{not\ c\}}$, expressing that contradiction is avoided by not assuming the optative hypothesis *not c*. It still allows the conclusions $\{\neg a, not\ a, b\}$.

8.3 Contradiction removal

It has argued in the introduction to this chapter that, to deal with the isuue of contradiction brought about by closed world assumptions, rather then defining more sceptical semantics one can rely instead on a less sceptical semantics and accompany it with a revision process that restores consistency, whenever violation of integrity contraints occurs.

In this section we define a revision process, that restores consistency for programs contradictory with respect to *WFSX*. This process relies on the allowing to take back assumptions about the truth of negative literals.

The set negative literals on which a revision can be made, i.e. the assumption of their truthfulness can be removed, is the set of *revisable literals*, and can be any subset of *not* $\mathcal{H}$.

In [Pereira *et al.*, 1991a] a revision semantics was defined where only base closed world assumption are revisables. There revisables are default literals whose complement has no rules. In [Pereira *et al.*, 1992b] the notion of base closed world assumption was improved, in order to deal with the case of loops without interposing *not* s[7]. The notion of revisables presented there is similar to the notion of prime optatives above.

As we show in section 8.4 the issue of which are the revisables (in contradiction removal) is tantamount to that of which are the optatives (in contradiction avoidance). Thus the discussion on primacy of optatives is applicable to the issue of what literals are to be revisables.

So no restriction is made here on which default literals should be considered revisables. Revisable literal are supposed provided by the user along with the program[8].

For instance, in example 8.2 the revisable literals might be:

$$\{not\ fd, not\ lv, not\ pt, not\ bs\}$$

By not introducing *not* fd in this set, we are declaring that, in order remove some contradiction, we will not consider directly revising its truth value. However, this does not mean that by revising some other literal the truth value of *not* fd will not changed.

We take back revisable assumptions, i.e. assumptions on revisable literals, in a minimal way, and in all alternative ways of removing contradiction. Moreover, we identify a single unique revision that defines a sceptical revision process which includes all alternative contradiction removing revisions, so as not to prefer one over the other. This is akin in spirit to the approach of PSMs in [Przymusinska and Przymusinski, 1990, Przymusinski, 1990a], where the WFM is the intersection of all the PSMs.

The notions of minimality and contradiction removal employed are useful for dealing with Belief Revision through *WFSX*. Consider the noncontradictory program P :

$$\begin{array}{rcl} p & \leftarrow & not\ q \\ \neg p & \leftarrow & r, not\ t \end{array}$$

and the additional information: r. Our proposed revision for $P \cup \{r\}$ provides the minimal model $\{r\}$, and two extended additional ones, namely:

$$\{r, p, not\ \neg p, not\ q\} \quad \text{and} \quad \{r, \neg p, not\ p, not\ t\}.$$

These two models can be seen as alternative minimal changes to the WFM of P in order to incorporate the new information: one making t undefined rather than false by CWA, and the

[7] If *not* a is considered a base closed world assumption in a program without rules for a, then there is no reason for *not* a not being one such assumption in a program where the only rule for a is $a \leftarrow a$.

[8] The declaration of revisable literals by the user is akin to that of abducible literals. Although some frameworks identify what are the abducible for some particular problems ([Eshghi and Kowalski, 1989] where abducibles are of the form a^*), theories of abduction, for the sake of generality, make no restriction on which literals are abducibles, and assume them provided by the user.

other making q undefined instead. Model $\{r\}$ is obtained by making both t and q undefined. It is the one with sufficient and necessary changes compatible with the new information, whenever no preference is enforced about which relevant revisable literals to unassume, in fact by unassuming them all. Revisions can be defined as those programs, obtained from the original one in a unique way, whose *WFSX* are each of the noncontradictory models above. In this example these programs are:

$$\begin{array}{lllll} P & \cup & \{r\} & \cup & \{t \leftarrow not\ t\} \\ P & \cup & \{r\} & \cup & \{q \leftarrow not\ q\} \\ P & \cup & \{r\} & \cup & \{t \leftarrow not\ t;\quad q \leftarrow not\ q\} \end{array}$$

Notice how a rule of the form $L \leftarrow not\ L$ changes the assumption $not\ L$ from true to undefined.

The structure of this section is as follows: first we present a paraconsistent extension of *WFSX*. Then we define the intended revisions declaratively. Afterwards we define some useful sets for establishing the causes of and the removal of contradictions within *WFSX*, and prove that the result of their use concurs with the intended revisions defined. Finally some hints for the implementation are given.

8.3.1 Paraconsistent *WFSX*

In order to revise possible contradictions we need first to identify those contradictory sets implied by a program under a paraconsistent *WFSX*. The main idea here is to compute all consequences of the program, even those leading to contradictions, as well as those arising from contradictions. The following example provides an intuitive preview of what we intend to capture:

Example 8.21 Consider program P :

$$\begin{array}{llll} a \leftarrow & not\ b & \text{(i)} & \qquad d \leftarrow \quad not\ a \quad \text{(iii)} \\ \neg a \leftarrow & not\ c & \text{(ii)} & \qquad e \leftarrow \quad not\ \neg a \quad \text{(iv)} \end{array}$$

1. $not\ b$ and $not\ c$ hold since there are no rules for either b or c
2. $\neg a$ and a hold from 1 and rules (i) and (ii)
3. $not\ a$ and $not\ \neg a$ hold from 2 and the coherence principle
4. d and e hold from 3 and rules (iii) and (iv)
5. $not\ d$ and $not\ e$ hold from 2 and rules (iii) and (iv), as they are the only rules for d and e
6. $not\ \neg d$ and $not\ \neg e$ hold from 4 and the coherence principle.

The whole set of literal consequences is then:

$$\{not\ b, not\ c, \neg a, a, not\ a, not\ \neg a, d, e, not\ d, not\ e, not\ \neg d, not\ \neg e\}.$$

Without loss of generality (cf. corollary 10.1.1), and for the sake of simplicity, we consider that programs are always in their canonical form (cf. definition 2.1.1).

For the purpose of defining a paraconsistent extension of *WFSX*, we begin by defining what an interpretation is in the paraconsistent case.

Definition 8.3.1 (p–interpretation) *A p–interpretation I is any set $T \cup not\ F$, such that if $\neg L \in T$ then $L \in F$ (coherence).*

The modification of the Coh operator is also straightforward:

Definition 8.3.2 (The Coh^p operator) *Let $QI = QT \cup not\ QF$ be a set of literals. We define $Coh^p(QI)$ as the p–interpretation $T \cup not\ F$ such that*

$$T = QT \text{ and } F = QF \cup \{\neg L \mid L \in T\}.$$

Note that in both definitions the enforcement of disjointness on sets T and F has been withdrawn.

Now we generalize the modulo transformation (definition 4.2.1 in page 31) to the paraconsistent case. If we assume, without loss of generality, that programs are always in their canonical form, according to theorem 4.2.4 the generalization can be made in the compact version of the transformation, thereby simplifying the exposition.

In the compact definition of the $\frac{P}{I}$ transformation one can apply the first two operations in any order, because the conditions of their application are disjoint for any interpretation. A potencial conflict would rest on applying both the first and the second operation, but that can never happen because if some $A \in I$ then $not\ A \notin I$, and vice–versa.

This is not the case for p–interpretations pI, where for some objective literal A both A and $not\ A$ might belong to pI. Thus if one applies the transformation to p–interpretations, different results are obtained depending on the order of the application of the first two operations.

Example 8.22 Consider program P of example 8.21, and let us compute:

$$\frac{P}{\{a, \neg a, not\ \neg a, not\ a, not\ b, not\ c\}}.$$

If one applies the operations in the order they are presented:

- Rules (iii) and (iv) of P are removed since both a and $\neg a$ belong to the p–interpretation.
- $not\ b$ and $not\ c$ are removed from the bodies of rules since $not\ b$ and $not\ c$ belong to the p–interpretation.

and the resulting program is:

$$\begin{array}{rl} a & \leftarrow \\ \neg a & \leftarrow \end{array}$$

But if one applies the second operation first:

- $not\ b$, $not\ c$, $not\ a$, and $not\ \neg a$ are removed from the bodies of rules since $not\ b$, $not\ c$, $not\ a$, and $not\ \neg a$ belong to the p–interpretation.
- Since no literals remain in the body of rules no other operation is applicable.

The resulting program in this case is:

$$\begin{array}{rlrl} a & \leftarrow & d & \leftarrow \\ \neg a & \leftarrow & e & \leftarrow \end{array}$$

In order make the transformation independent of the order of application of the operations we define the corresponding transformation for the paraconsistent case as being nondeterministic in the order of application of those rules.

Definition 8.3.3 ($\frac{P}{I}p$ transformation) *Let P be a canonical extended logic program and let I be a p–interpretation. By a $\frac{P}{I}p$ program we mean any program obtained from P by first non–deterministically applying the operations until they are no longer applicable:*

- *Remove all rules containing a default literal $L = not\ A$ such that $A \in I$.*

- *Remove from rules their default literals $L = not\ A$ such that $not\ A \in I$.*

and by next replacing all remaining default literals by proposition **u**.

In order to get all consequences of the program, even those leading to contradictions, as well as those arising from contradictions, we consider the consequences off all possible such $\frac{P}{I}p$ programs.

Definition 8.3.4 (The Φ^p operator) *Let P be a canonical extended logic program, I a p–interpretation, and let P_k such that $k \in K$ be all the possible results of $\frac{P}{I}p$. Then:*

$$\Phi^p{}_P(I) = \bigcup_{k \in K} Coh^p(least(P_k))$$

Theorem 8.3.1 (Monotonicity of Φ^p) *The Φ^p operator is monotonic under set inclusion of p–interpretations.*

Proof: We have to prove that for any two p–interpretation A and B such that $A \subseteq B$, then $\Phi^p(A) \subseteq \Phi^p(B)$.

Let $P_{A_k},\ k \in K$, and $P_{B_j},\ j \in J$, be the programs obtained from, respectively, $\frac{P}{A}p$ and $\frac{P}{B}p$. Since $A \subseteq B$ then for every P_{A_k} there exists a P_{B_j} such that for every rule

$$H \leftarrow Body \in P_{B_j}$$

there exists a rule

$$H \leftarrow Body \cup Body' \in P_{A_k}.$$

This is necessarily the case because B, having more literals than A, can always remove more rules and default literals in the bodies than A. Thus:

$$\forall P_{A_k}\ \exists P_{B_j}\ \mid\ least(P_{A_k}) \subseteq least(P_{B_j})$$

Now we prove that Coh^p is also monotonic, i.e for any two p–interpretations

$$I = T_I \cup not\ F_I \text{ and } J = T_J \cup not\ F_J$$

such that

$$T_I \subseteq T_J \text{ and } F_I \subseteq F_J,$$

$Coh^p(I) \subseteq Coh^p(J)$ holds.

$Coh^p(I) \subseteq Coh^p(J)$ is equivalent, by definition of Coh^p, to

$$T_I \cup not\ (F_I \cup \{\neg L \mid L \in T_I\}) \subseteq T_J \cup not\ (F_J \cup \{\neg L \mid L \in T_J\})$$

since $T_I \subseteq T_J$ by hypothesis, the above is true if:

$$F_I \cup \{\neg L \mid L \in T_I\} \subseteq F_J \cup \{\neg L \mid L \in T_I\} \cup \{\neg L \mid L \in T_J - T_I\}$$

which is equivalent to

$$F_I \subseteq F_J \cup \{\neg L \mid L \in T_J - T_I\}$$

which holds because, by hypothesis, $F_I \subseteq F_J$.

With this result, and the other one above:

$$\forall P_{A_k}\ \exists P_{B_j}\ \mid\ Coh^p(least(P_{A_k})) \subseteq Coh^p(least(P_{B_j}))$$

and consequently:

$$\bigcup_{k\in K} Coh^p(least(P_{A_k})) \subseteq \bigcup_{j\in J} Coh^p(least(P_{B_j}))$$

$\diamondsuit$

Given that Φ^p is monotonic, then for every program it always has a least fixpoint, and this fixpoint can be obtained by iterating Φ^p starting from the empty set:

Definition 8.3.5 (Paraconsistent *WFSX*) *The paraconsistent* WFSX *of an (canonical) extended logic program P, denoted by* $\text{WFSX}_p(P)$*, is the least fixpoint of Φ^p applied to P.*

If some literal L belongs to the paraconsistent WFSX *of P we write:*

$$P \models_p L$$

Proposition 8.3.1 (Existence of *WFSX*$_p$) $\text{WFSX}_p(P)$ *is defined for every program with ICs.*

Proof: Since no restriction whatsoever has been made on the application of Φ^p, and given the proof of monotonicity of this operator, a least fixpoint of it exists for every program. $\diamondsuit$

Example 8.23 Let us compute the paraconsistent *WFSX* of the program in example 8.21. P is already in canonical form.

We start with the empty set. The only program obtained from $\frac{P}{\{\}}p$ is $P_{0,1}$:

$$\begin{array}{rcl} a & \leftarrow & \mathbf{u} \\ \neg a & \leftarrow & \mathbf{u} \end{array} \qquad \begin{array}{rcl} d & \leftarrow & \mathbf{u} \\ e & \leftarrow & \mathbf{u} \end{array}$$

and $I_1 = Coh^p(least(P_{0,1})) = \{not\ b, not\ c\}$

By $\frac{P}{I_1}p$ we only get one program, $P_{1,1}$:

$$\begin{array}{rcl} a & \leftarrow & \\ \neg a & \leftarrow & \end{array} \qquad \begin{array}{rcl} d & \leftarrow & \mathbf{u} \\ e & \leftarrow & \mathbf{u} \end{array}$$

and $I_2 = Coh^p(least(P_{1,1})) = \{a, not\ \neg a, \neg a, not\ a, not\ b, not\ c\}$

The result of $\frac{P}{I_2}p$ are the four programs:

$$\begin{array}{rcl} P_{2,1}: & & \\ a & \leftarrow & \\ \neg a & \leftarrow & \\ d & \leftarrow & \\ e & \leftarrow & \end{array} \qquad \begin{array}{rcl} P_{2,2}: & & \\ a & \leftarrow & \\ \neg a & \leftarrow & \\ d & \leftarrow & \end{array} \qquad \begin{array}{rcl} P_{2,3}: & & \\ a & \leftarrow & \\ \neg a & \leftarrow & \\ e & \leftarrow & \end{array} \qquad \begin{array}{rcl} P_{2,4}: & & \\ a & \leftarrow & \\ \neg a & \leftarrow & \end{array}$$

For example, $P_{2,1}$ was obtained by applying the second operation to both rules (iii) and (iv), which is possible because both *not* a and *not* $\neg a$ belong to I_2. $P_{2,4}$ was obtained by applying the first operation to both rules (iii) and (iv), which is possible because both a and $\neg a$ belong to I_2.

It is easy to see that $I_3 = \Phi^p(I_2) =$

$$\{not\ b, not\ c, \neg a, a, not\ a, not\ \neg a, d, e, not\ d, not\ e, not\ \neg d, not\ \neg e\}$$

By applying $\frac{P}{I_3}p$ one gets exactly the same program as in $\frac{P}{I_2}p$ and thus $\Phi^p(I_3) = I_3$. So, I_3 is the least fixpoint of Φ^p and, consequently, the paraconsistent *WFSX* of P.

Now we can give a definition of a contradictory program with ICs:

Definition 8.3.6 (Contradictory program with ICs) *A program P with language Lang where A is an atom, and a set of integrity constraints IC is contradictory iff*

$$P \cup ICs \cup \{\bot \leftarrow A, \neg A \mid A \in Lang\} \models_p \bot$$

In this section we always refer to the paraconsistent *WFSX* as an extension of *WFSX* for noncontradictory programs. This is so because:

Proposition 8.3.2 *For a noncontradictory program P the paraconsistent* WFSX *coincides with* WFSX.

Proof: Since interpretations are p–interpretations, and for any noncontradictory set S of literals $Coh(S) = Coh^p(S)$, and for any interpretation I $\frac{P}{I}p$ is deterministic and equal to $\frac{P}{I}$, the result follows trivially. $\diamondsuit$

8.3.2 Declarative revisions

Before tackling the question of which assumptions to revise to abolish contradiction, we begin by showing how to impose in a program a revision that takes back some revisable assumption, identifying rules of a special form, which have the effect of prohibiting the falsity of an objective literal in models of a program. Such rules can prevent an objective literal being false, hence their name:

Definition 8.3.7 (Inhibition rule) *The inhibition rule for a default literal not L is:*

$$L \leftarrow not\ L$$

By $IR(S)$ where S is a set of default literals, we mean:

$$IR(S) = \{L \leftarrow not\ L \mid not\ L \in S\}$$

These rules state that if *not* A is true then A is also true, and so a contradiction arises. Intuitively this is quite similar to the effect of integrity constraints of form $\bot \leftarrow not\ A$. Technically the difference is that the removal of such a contradiction in the case of inhibition rules is dealt by *WFSX*itself, where in the case of those integrity constraints isn't.

Proposition 8.3.3 *Let P be any program such that for objective literal L, $P \not\models_p \neg L$. Then:*

$$P \cup \{L \leftarrow not\ L\} \not\models_p not\ L$$

Moreover, if there are no other rules for L, the truth value of L is undefined in $\text{WFSX}_p(P)$.

Proof: Let $P' = P \cup \{L \leftarrow not\ L\}$. We prove by transfinite induction that:

$$not\ L \notin I_\alpha, \quad \text{where } I_\alpha = \Phi^{p\uparrow\alpha}(\{\})$$

- *For limit ordinals:* Since $\Phi^{p\uparrow 0}(\{\}) = \{\}$, $not\ L \notin I_0$.
  For limit ordinal δ, suppose that for all $\alpha < \delta$

  $$not\ L \notin \Phi^{p\uparrow\alpha}(\{\})$$

  Then, clearly:

  $$not\ L \notin \bigcup \left\{\Phi^{p\uparrow\alpha}(\{\}) \mid \alpha < \delta\right\}$$

  i.e. $not\ L \notin \Phi^{p\uparrow\delta}(\{\})$.

- *Induction step* Assume that $not\ L \notin I_i$, for some ordinal i. Then:
  - if $L \notin I_i$ then every transformed program $\frac{P'}{I_i}p$ has the rule $L \leftarrow \mathbf{u}$. Thus for every transformed program
    $$not\ L \notin least\left(\frac{P'}{I_i}p\right)$$
    and given that by hypothesis $P \not\models_p \neg L$
    $$not\ L \notin Coh^p\left(least\left(\frac{P'}{I_i}p\right)\right).$$
    Thus $not\ L \notin I_{i+1}$.
  - if Ł $\in I_i$ then by monotonicity of Φ^p every transformed program has a rule $L \leftarrow$, and thus $not\ L \notin I_{i+1}$.

Since $WFSX_p(P') = \Phi^{p\uparrow\lambda}(\{\})$ for some smallest ordinal λ, then $not\ L \notin WFSX_p(P')$. $\diamondsuit$

These rules allows, by adding them to a program, to force default literals in the paraconsistent *WFSX* to become undefined. Note that changing the truth value of revisable literals from true to undefined is less committing than changing it to false. In order to obtain revisions where the truth value of revisable literals is changed from true to false, one has to iterate the process we're about to define. The formal definition of such revisions can be found in [Pereira *et al.*, 1993d].

To declaratively define the intended program revisions void of contradiction we start by first considering the resulting *WFSX*s of all possible ways of revising a program P with inhibition rules, by taking back revisable assumptions, even if some revisions are still contradictory programs.

However, it might happen that several different revisions in fact correspond to the same, in the sense that they lead to the same consequences.

Example 8.24 Consider program P :

$$\begin{array}{lcl} \bot & \leftarrow & not\ a \\ \\ a & \leftarrow & b \\ b & \leftarrow & a \\ a & \leftarrow & c \end{array}$$

with revisables $Rev = \{not\ a, not\ b, not\ c\}$.

Note that adding $a \leftarrow not\ a$, $b \leftarrow not\ b$, or both, leads to the same consequences. Intuitively they are the same revision, since undefining a leads to the undefinedness of b and vice–versa. Considering all three as distinct can be misleading because it appear that the program has three differente revisions.

Revisables $not\ a$ and $not\ b$ are indissociable, and it is indifferent to introduce inhibition rules for one, the other, or both. Moreover, only one of these hypotheses should be considered as a revision. In the sequel, we coalesce the three revisions into a single standard one, that adds both inhibition rules.

Definition 8.3.8 (Indissociable literals) *Let P be an extended logic program with revisables Rev. The set $Ind(S) \supseteq S$ of indissociable literals of a set S of default literals is the largest subset of Rev such that:*

- $Ind(S) \subseteq \mathrm{WFSX}_p(P)$ *and*

- $\text{WFSX}_p(P \cup IR(S)) \cap Ind(S) = \{\}$

i.e. $Ind(S)$ is the set of all revisables that change their truth value from true to undefined, once inhibition rules are added for every default literals of S to change their truth value.

It is easy to see that such a largest set always exists (since Ind is monotonic), and that Ind is a closure operator. Moreover:

Proposition 8.3.4 *Let $M = \text{WFSX}_p(P \cup IR(S))$ for some subset of S of Rev. Then:*

$$\text{WFSX}_p(P \cup IR(Ind(S))) = M$$

Proof: Let $P' = P \cup IR(S)$, and let *not* L be an arbitrary literal such that *not* $L \in Ind(S)$ and *not* $L \notin S$.

Directly from the second point of the definition of indissociables, it follows that *not* L is undefined in P'. Moreover, it is clear that the addition into any program P, of an inhibition rule for some literal undefined in $WFSX_p(P)$ does not change the well–founded model. Thus:

$$WFSX_p(P') = WFSX_p(P' \cup IR(\{not\ L\}))$$

$\diamondsuit$

Example 8.25 In example 8.24:

$$Ind(\{not\ a\}) = Ind(\{not\ b\}) = \{not\ a, not\ b\}$$

and

$$Ind(\{not\ c\}) = \{not\ a, not\ b, not\ c\}$$

Definition 8.3.9 (Submodels of a program) *A submodel of a (contradictory) program P with ICs, and revisable literals Rev, is any pair $\langle M, R\rangle$ where R is a subset of Rev closed under indissociable literals, i.e:*

$$\forall S \subseteq R,\ Ind(S) \subseteq R$$

and:

$$M = \text{WFSX}_p(P \cup \{L \leftarrow not\ L \mid not\ L \in R\})^9.$$

In a submodel $\langle M, R\rangle$ we dub R the submodel revision, and M are the consequences of the submodel revision. A submodel is contradictory iff M is contradictory (i.e. either contains $\bot$ or is not an interpretation)[10].

The existence of $WFSX_p(P)$ for any program P (cf. proposition 8.3.1) grants that M exists for every subset of Rev. Moreover, since Ind is a closure operator:

Proposition 8.3.5 (Submodels lattice) *The set of all submodels $\langle M, R\rangle$ of any program P with revisable literals Rev forms a complete lattice under set inclusion on the submodel revisions.*

The submodels lattice of example 8.24 is presented in figure 8.1.

Example 8.26 Consider program P :

$$\begin{array}{rcl} p & \leftarrow & not\ q \\ \neg p & \leftarrow & not\ r \\ a & \leftarrow & not\ b \end{array}$$

with revisable literals $Rev = \{not\ q, not\ r, not\ b\}$. Its submodels lattice is depicted in figure 8.2, where shadowed submodels are contradictory ones. For simplicity, contradictory models are not presented in full in the figure.

[9] For a study of submodels based on the PSMs instead of on the well–founded model see [Pereira *et al.*, 1991b].

[10] Note the one–to–one correspondence between submodels and program revisions.

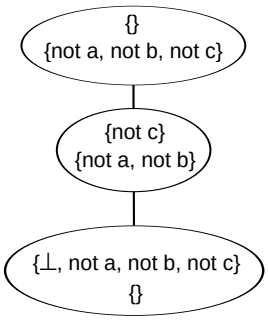

Figure 8.1: Submodels lattice of example 8.24.

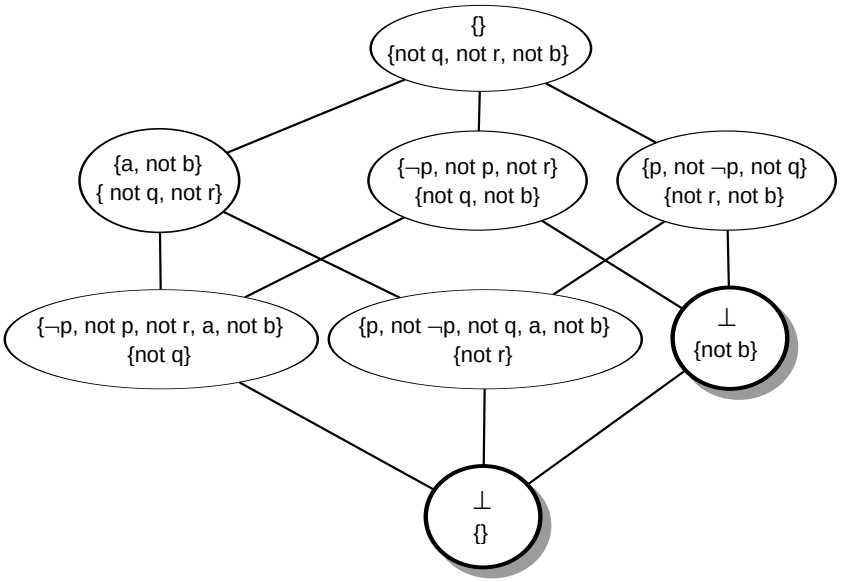

Figure 8.2: Submodels lattice of example 8.26

As we are interested in revising contradiction in a minimal way, we care about those submodels that are noncontradictory and among these, about those that are minimal in the submodels lattice.

Definition 8.3.10 (Minimal noncontradictory submodel) *A submodel $\langle M, R\rangle$ is a minimal noncontradictory submodel (MNS for short) of a program P iff it is noncontradictory and there exists no other noncontradictory submodel $\langle M', R'\rangle$, such that $R' \subset R$.*

By definition, each MNS of a program P reflects a revision of P, $P \cup RevRules$[11] that guarantees noncontradiction, and such that for any set of rules $RevRules' \subseteq RevRules$ closed under indissociables, $P \cup RevRules'$ is contradictory. In other words, each MNS reflects a revision of the program that restores consistency, and which adds a minimal set, closed under indissociables, of inhibition rules for revisables.

Example 8.27 Consider program P :

$$\begin{array}{rcl} p(X) & \leftarrow & p(s(X)) \\ a & \leftarrow & not\ p(s(X)) \\ \neg a & & \end{array}$$

where $s(X)$ denotes the successor of X, and let $Rev = \{not\ p(i) \mid i > 0\}$.

The only sets of inhibition rules that remove the contradiction are $IR(S_k)$, such that:

$$S_k = \{not\ p(i) \mid i > k\}.$$

None of them is minimal.

However the closure under indissociable of each of them is:

$$S = \{not\ p(i) \mid i > 0\}$$

Thus the only noncontradictory submodel is $\langle M, S\rangle$, where

$$M = \{\neg a, not\ a, not\ p(0)\}$$

and so it is also the only MNS.

Note that the revision models of each of the revisions above is indeed M (cf. proposition 8.3.4).

It is also clear that literals in the submodel revision indeed change their truth value once the inhibition rules are added:

Proposition 8.3.6 *If $\langle M, R\rangle$ is a MNS of program P then:*

$$R \subseteq \mathrm{WFSX}_p(P)$$

Proof: Assume the contrary, i.e. $\langle M, R\rangle$ is a MNS of P and $R \not\subseteq WFSX_p(P)$. Then:

$$\exists not\ L \in R \mid not\ L \notin WFSX_p(P)$$

Thus, the addition of inhibition rule $L \leftarrow not\ L$ has no effect in $WFSX_p(P)$. Consequently, $R - \{not\ L\}$ is a noncontradictory submodel of P, and so $\langle M, R\rangle$ is not minimal. $\diamondsuit$

[11] Where *RevRules* is the set of inhibition rules for some submodel revision.

Definition 8.3.11 (Minimally revised program) *Let P be a program with revisable literals Rev, and $\langle M, R\rangle$ some MNS of P. A minimally revised program MRP of P is:*

$$P \cup IR(R)$$

i.e. P plus one inhibition rule for each element of R.

It is clear that:

Proposition 8.3.7 *If P is noncontradictory its single MNS is*

$$\langle \mathrm{WFSX}(P), \{\}\rangle,$$

and P itself is its only minimally revised program

Example 8.28 The minimally revised programs of the program in example 8.26 are:

$$\begin{array}{lcl} MRP_1 & = & \{p \leftarrow not\ q; \neg p \leftarrow not\ r; a \leftarrow not\ b; q \leftarrow not\ q\} \quad and \\ MRP_2 & = & \{p \leftarrow not\ q; \neg p \leftarrow not\ r; a \leftarrow not\ b; r \leftarrow not\ r\}. \end{array}$$

Each of these two programs is a transformation of the original one that minimally removes contradiction by taking back the assumption of truth of some revisables via their inhibition rules[12]. In this example, one can remove the contradiction in p either by going back on the closed world assumption of falsity of q (or truth of *not* q) or on the falsity of r. The program that has the first effect is MRP_1, the one with the second effect being MRP_2. Having no reason to render q alone, or r alone undefined, it is natural that a sceptical revision should accomplish the effect of undefining them both.

Definition 8.3.12 (Sceptical revision) *The sceptical submodel of a program P is the join $\langle M_J, R_J\rangle$ of all MNSs of P. The sceptical revised program of P is the program obtained from P by adding to it an inhibition rule for each element of R_J.*

It is important to guarantee that the sceptical revision indeed removes contradiction from a program. This is so because:

Proposition 8.3.8 *Let $\langle M_1, R_1\rangle$ and $\langle M_2, R_2\rangle$ be any two noncontradictory submodels. Then submodel $\langle M, R_1 \cup R_2\rangle$ is also noncontradictory.*

Proof: Since it is clear that $R_1 \cup R_2$ is closed under indissociable, we only have to prove that $\bot \notin M$. Since $\bot \notin M_1$ and $\bot \notin M_2$ it is enough to prove that $M \subseteq M_1 \cap M_2$.

By definition:

$$M = WFSX_p(P \cup \{L \leftarrow not\ L \mid not\ L \in R1 \cup R2\}).$$

As the extra rules only make literals undefined, and undefinedness only results in undefinedness, adding them all leads at most to the same set of literals being true or false, compared to adding them separately for only R_1 or R_2. $\diamondsuit$

Example 8.29 Consider contradictory program P :

$$\begin{array}{lcllcl} p & \leftarrow & not\ q & \neg p & \leftarrow & not\ a \\ q & \leftarrow & not\ r & \neg a & \leftarrow & not\ b \\ r & \leftarrow & not\ s & & & \end{array}$$

with revisables $Rev = \{not\ q, not\ a, not\ b\}$. Figure 8.4 shows its submodels lattice, where MNSs are shadowed and the sceptical submodels is in bold.

[12]Non–minimally revised programs can be defined similarly, by considering all noncontradictory submodels instead of minimal ones only. We won't consider them in the sequel however, though they are useful for other purposes: viz. counterfactual reasoning, as defined in [Pereira *et al.*, 1991d].

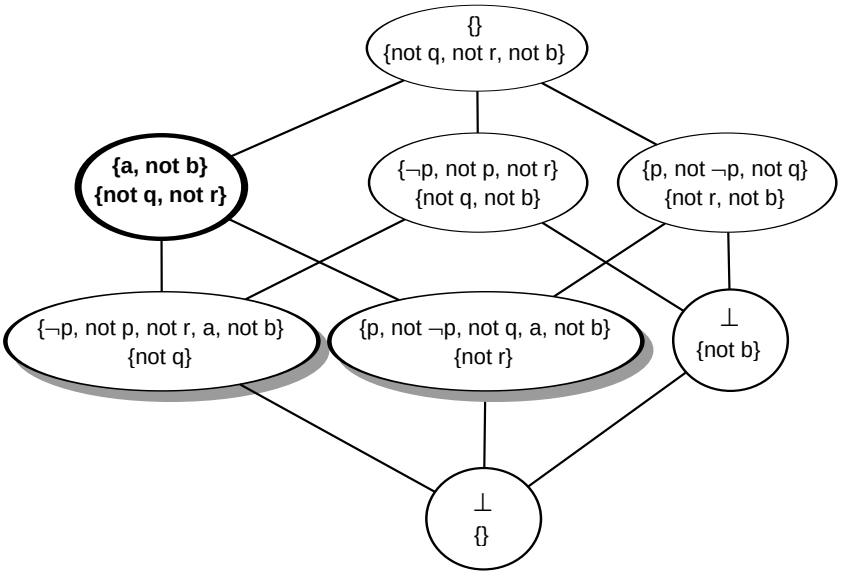

Figure 8.3: The MNSs of the program from example 8.26 are shadowed. Its sceptical submodel, the join of the MNSs, is in bold. Note that inhibiting b is irrelevant for revising P, and how taking the join of the MNSs captures what's required.

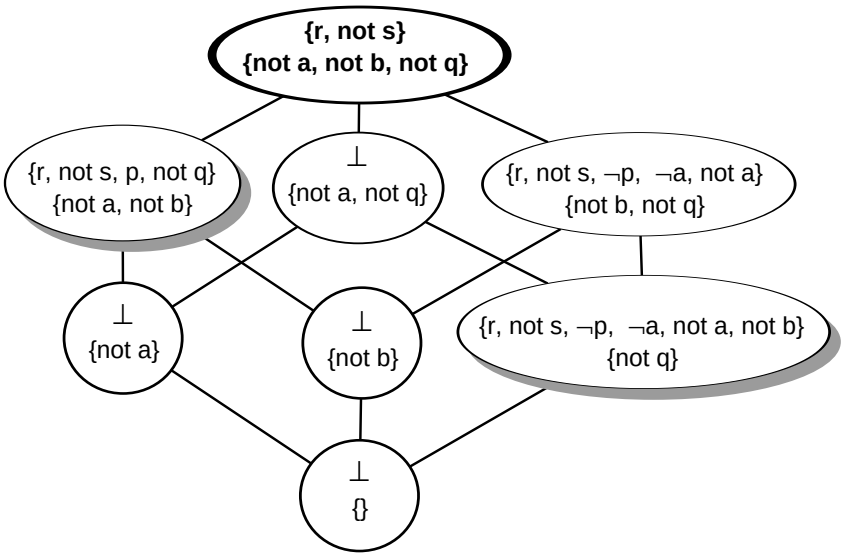

Figure 8.4: Sceptical submodels and MNSs of example 8.29.

Example 8.30 Consider the so-called Nixon diamond:

$$\begin{array}{rcl} \bot & \leftarrow & pacifist(X), hawk(X) \\ pacifist(X) & \leftarrow & quaker(X), not\ ab\_quaker(X) \\ hawk(X) & \leftarrow & republican(X), not\ ab\_republican(X) \\ quaker(nixon) & & \\ republican(nixon) & & \end{array}$$

This contradictory program P has two MRPs:

- one by adding to P $ab\_quaker \leftarrow not\ ab\_quaker$
- another by adding to P $ab\_republican \leftarrow not\ ab\_republican$

Both these programs have noncontradictory *WFSX*s:

$$\{hawk(nixon), quaker(nixon), republican(nixon), \\ not\ ab\_republican(nixon)\}$$

$$\{pacifist(nixon), quaker(nixon), republican(nixon), \\ not\ ab\_quaker(nixon)\}$$

The sceptical submodel of the program results from adding to it both inhibition rules. Its *WFSX* is $\{quaker(nixon), republican(nixon)\}$.

The importance of having a single sceptical revision can be observed here, since there is no reason for prefering between Nixon being a pacifist or a hawk. Nevertheless, the other revisions also give relevant information[13].

It is clear that with these intended revisions some programs have no revision. This happens when contradiction has a basis on non–revisable literals.

Example 8.31 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ \neg a & & \end{array} \qquad \begin{array}{rcl} b & \leftarrow & not\ c \\ c & & \end{array}$$

with revisables $Rev = \{not\ c\}$.

The only submodels of P are:

$$\langle WFSX_p(P), \{\}\rangle \text{ and } \langle WFSX_p(P \cup \{c \leftarrow not\ c\}), \{not\ c\}\rangle.$$

As both these submodels are contradictory P has no MNS, and thus no revisions. Note that if $not\ b$ were revisable,the program would have a revision $P \cup \{b \leftarrow not\ b\}$. If $not\ b$ were absent from the first rule, P would have no revision no matter what revisables.

Definition 8.3.13 (Unrevisable program) *A contradictory program P with revisables Rev is unrevisable iff it has no noncontradictory submodel.*

However it is possible to guarantee that consistent programs have revisions.

Proposition 8.3.9 *Let P be a consistent program with ICs and revisable literals $Rev = not\ \mathcal{H}$. Then, if P is contradictory it is revisable.*

Proof: By definition of consistent program (definition 8.2.2), if no negative literal is assumed, the program is noncontradictory. Thus, at least the submodel obtained by adding to P an inhibition rule for every objective literal L in $\mathcal{H}$ such that $P \not\models_p \neg L$, is noncontradictory. $\diamondsuit$

[13]Their *WFSX*s correspond to the two usual default extensions.

8.3.3 Contradiction support and removal

Submodels characterize which are the possible revisions, and the minimality criterium. Of course, a procedure for finding the minimal and the sceptical submodels can hardly be based on their declarative definition: one have to generate all the possible revisions to select these intended ones. In this section we define a revision procedure, and show that it concurs with the declaratively intended revisions.

The procedure relies on the notions of contradiction support, and of contradiction removal sets. Informally, contradiction supports are sets of revisable literals present in the $WFSX_p$ which are sufficient to support $\bot$ (i.e. contradiction)[14]. From their truth the truth of $\bot$ inevitably follows.

Contradiction removal sets are built from the contradiction supports. They are minimal sets of literals chosen from the supports such that any support of $\bot$ contains at least one literal in the removal set. Consequently, if all literals in some contradiction removal set were to become undefined in value then no support of $\bot$ would subsist. Thus removal sets are the hitting sets of the supports.

Example 8.32 Consider the program of example 8.26. Its only contradiction support is $\{not\ q, not\ r\}$, and its contradiction removal sets are $\{not\ q\}$ and $\{not\ r\}$.

Suppose we had q undefined as a result of rules for q. In that case $\bot$ would also be undefined, the program becoming noncontradictory. The same would happen if r alone were undefined. No other set, not containing one of these two alternatives, has this property.

Definition 8.3.14 (Support of a literal) *The supports*[15] *of a literal L belonging to* WFSX_p *of a program P with revisables Rev (each represented as $SS(L)$) are obtained as follows:*

1. *If L is an objective literal:*

   (a) *If there is a fact for L then a support of L is $SS(L) = \{\}$.*

   (b) *For each rule:*

   $$L \leftarrow B_1, \ldots, B_n \qquad n \geq 1$$

   in P such that $\{B_1, \ldots, B_n\} \subseteq \mathrm{WFSX}_p(P)$, there exists a support of L

   $$SS(L) = \bigcup_i SS_{j(i)}(B_i)$$

   for each combination of one $j(i)$ for each i.

2. *If $L = not\ A$ (where A is an objective literal):*

   (a) *If $L \in Rev$ then a support of L is $SS(L) = \{L\}$.*

   (b) *If $L \notin Rev$ and there are no rules for A then a support of L is $SS(L) = \{\}$.*

   (c) *If $L \notin Rev$ and there are rules for A, choose from each rule defined for A, a literal such that its default complement belongs to* $\mathrm{WFSX}_p(P)$*. For each such choice there exists several $SS(L)$; each contains one support of each default complement of the choosen literals.*

[14] This notion can be seen as a special case of the notion of Suspect Sets introduced in declarative debugging in [Pereira and Calejo, 1988].

[15] An alternative definition of supports relies on a notion of derivation for a literal in the $WFSX_p$, and doesn't require the previous availability of the WF Model. This is, however, beyond the scope of this work. For derivation procedures for $WFSX_p$ the reader is referred to [Aparício, 1993].

(d) If $\neg A \in \text{WFSX}_p(P)$ *then there are, additionally, supports*

$$SS(L) = SS_k(\neg A)$$

for each k.

Example 8.33 Consider program P of example 8.29, whose paraconsistent well–founded consequences are:

$$WFSX_p(P) = \{not\ s, r, not\ q, p, not\ \neg p, not\ b, \neg a, not\ a, \neg p, not\ p\}$$

The supports of p are computed as follows:

- From the only rule for p conclude that the supports of p are the supports of $not\ q$.
- Since $not\ q$ is a revisable then one of its supports is $\{not\ q\}$.
- As $\neg q \notin WFSX_p(P)$, there are no other supports of q.

Thus the only support of p is $\{not\ q\}$.

The supports of $\neg p$ are:

- From the only rule for $\neg p$ conclude that the supports of $\neg p$ are the supports of $not\ a$.
- Since $not\ a$ is a revisable then one of its support is $\{not\ a\}$.
- Since $\neg a \in WFSX_p(P)$, then supports of $\neg a$ are also supports of $not\ a$.
- From the only rule for $\neg a$ conclude that the supports of $\neg a$ are the supports of $not\ b$.
- Identically to $not\ q$ above, the only support of $not\ b$ is $\{not\ b\}$.

Thus $\neg p$ has two supports, namely $\{not\ a\}$ and $\{not\ b\}$.

Example 8.34 The supports of a in example 8.27 are:

$$\begin{aligned} SS_1(a) &= \{not\ p(1)\} \\ &\vdots \\ SS_i(a) &= \{not\ p(i)\} \\ &\vdots \end{aligned}$$

Proposition 8.3.10 (Existence of support) *A literal* L *belongs to the* WFSX_p *of a program* P *iff it has at least one support* $SS(L)$.

Proof: The proof follows directly from the results in [Aparício, 1993] regarding derivation procedures for $WFSX_p$. $\diamondsuit$

Definition 8.3.15 (Contradiction support) *A contradiction support of a program* P *is a support of* $\bot$ *in the program obtained from* P *by adding to it constraints of the form* $\bot \leftarrow L, \neg L$ *for every objective literal* L *in the language of* P.

N.B. From now on, unless otherwise stated, when we refer to a program we mean the program obtained by adding to it all such constraints.

Example 8.35 The contradiction supports of program P from example 8.29 are the union of pairs of supports of p and $\neg p$.

Thus, according to the supports calculated in example 8.33, P has two contradiction supports, namely $\{not\ q, not\ a\}$ and $\{not\ q, not\ b\}$.

Contradiction supports are sets of revisables true in the $WFSX_p$ of the program and involved in some support of contradiction (i.e. $\bot$)[16].

Having defined the sets of revisables that together support some literal, it is easy to produce sets of revisables such that, if all become undefined, the truth of that literal would necessarily become ungrounded. To coupe with indissociability, these sets are closed under indissociable literals.

Definition 8.3.16 (Removal set) *A pre–removal set of a literal L belonging to the* WFSX_p *of a program P is a set of literals formed by the union of some nonempty subset from each $SS(L)$.*

A removal set (RS) of L is the closure under indissociable literals of a pre–removal set of L.

If the empty set is a $SS(L)$, then the only $RS(L)$ is, by definition, the empty set. Note that a literal not belonging to $\mathrm{WFSX}_p(P)$ *has no RSs defined for it.*

In view of considering minimal changes to the WF Model, we next define those RSs which are minimal in the sense that there is no other RS contained in them.

Definition 8.3.17 (Minimal removal set) *In a program P, $RS_m(L)$ is minimal removal set iff there exists no $RS_i(L)$ in P such that*

$$RS_m(L) \supset RS_i(L).$$

We represent a minimal RS of L in P as $MRS_P(L)$.

Definition 8.3.18 (Contradiction removal set) *A contradiction removal set (CRS) of program P is a minimal removal set of the (reserved) literal $\bot$, i.e. a CRS of P is a $MRS_P(\bot)$.*

Example 8.36 Consider program P of example 8.24. The only support of $\bot$ is $SS(\bot) = \{not\ a\}$. Thus the only pre–removal set of $\bot$ is also $\{not\ a\}$. Since

$$Ind(\{not\ a\}) = \{not\ b\},$$

the only contradiction removal set is $\{not\ a, not\ b\}$.

Example 8.37 The removal sets of $\bot$ in the program of example 8.29 are:

$$\begin{array}{lcllcl} RS_1 & = & \{not\ q\} & RS_2 & = & \{not\ q, not\ a\} \\ RS_3 & = & \{not\ q, not\ b\} & RS_4 & = & \{not\ a, not\ b\} \end{array}$$

Thus RS_1 and RS_4 are contradiction removal sets. Note that these correspond exactly to the revisions of minimal noncontradictory submodels of figure 8.4.

Example 8.38 The only CRS of example 8.27 is:

$$CRS = \{not\ p(i) \mid i > 0\}$$

It is important to guarantee that contradiction removal sets do indeed remove contradiction.

Lemma 8.3.2 *Let P be a contradictory program with contradiction removal set CRS. Then:*

$$P \cup IR(CRS)$$

is noncontradictory.

[16]Note that there is a close relationship between the SSs of $\bot$ and the sets of nogoods of Truth Maintenance Systems.

Proof: By construction of removal set of $\bot$,

$$P' = P \cup \{L \leftarrow not\ L \mid not\ L \in CRS\}$$

has no support of $\bot$. Thus, by proposition 8.3.10, $\bot \notin WFSX_p(P')$. $\diamondsuit$

Now we prove that this process concurs with the intended revisions above. This is achieved by proving three theorems:

Theorem 8.3.3 (Soundness of CRSs) *Let R be a nonempty CRS of a contradictory program P. Then $\langle M, R\rangle$ is a MNS of P, where:*

$$M = \text{WFSX}(P \cup IR(R))$$

Proof: Since by definition R is closed under indissociables, it is clear that $\langle M, R\rangle$ is a submodel of P. By lemma 8.3.2, it is also a noncontradictory submodel of P.

Now, we prove, by contradiction, that there exists no noncontradictory submodel of P smaller than $\langle M, R\rangle$.

Let $\langle M', R'\rangle$ be a noncontradictory submodel, such that $R' \subset R$. If R' is not closed under indissociables, then $\langle M', R'\rangle$ is not a submodel of P. Otherwise, by construction of minimal removal sets $\bot$ has at least one support in the program obtained from P by introducing inhibition rules for elements of R'. Thus, by proposition 8.3.10, $\langle M', R'\rangle$ is a contradictory submodel. $\diamondsuit$

Theorem 8.3.4 (Completeness of CRSs) *Let $\langle M, R\rangle$ be a MNS, with $R \neq \{\}$, of a contradictory program P. Then R is a CRS of P.*

Proof: By proposition 8.3.6 $R \subseteq WFSX_p(P)$. So, by proposition 8.3.10, every literal of R has at least one support in P.

We begin by proving, by contradiction, that:

$$\forall not\ L \in R,\ \exists SS(\bot) \mid not\ L \in Ind(SS(\bot))$$

Assume the contrary. Then there exists a $not\ L \in R$ not belonging to the indissociables of any support of $\bot$. Thus, by definition of support, the supports of $\bot$ do not change if $L \leftarrow not\ L$ is added to P. Consequently:

$$\langle WFSX(P \cup IR(R - \{not\ L\})), R - \{not\ L\}\rangle$$

is a noncontradictory submodel of P, and so $\langle M, R\rangle$ is not minimal.

The rest of the proof follows by construction of removal sets, and its closure under indissociables. $\diamondsuit$

Theorem 8.3.5 (Unrevisable programs) *If $\{\}$ is a CRS of a program P then P is unrevisable.*

Proof: By definition, $\{\}$ can only be a CRS if it is a support of $\bot$. Note that in the calculus of $\{\}$ as a support of $\bot$, no rules for any of the revisables were taken into account. Thus if one adds inhibition rules for any combination of the revisables, $\{\}$ remains as a support of $\bot$ in any of the resulting programs. By proposition 8.3.10, $\bot$ belongs to the well–founded model of each of those programs, and so every submodel of P is contradictory. $\diamondsuit$

Theorem 8.3.6 (Sceptical revised program) *Let P be a contradictory program with CRSs, R_k such that $k \in K$. The sceptical revised program of P is:*

$$P \cup \left\{ L \leftarrow not\ L \mid \ not\ L \in \bigcup_{k \in K} R_i \right\}$$

Proof: The proof follows directly from theorems 8.3.3 and 8.3.4. $\diamondsuit$

Thus in order to compute the minimal and sceptical submodels:

- One starts by computing all supports of $\bot$. Although the definition of support requires one to know a priori the paraconsistent *WFSX*, an alternative definition exists such that this is not required. This definition is based on a top–down derivation procedure, which is beyond the scope of this work. Computing all supports of $\bot$ is like computing all the derivations for $\bot$ in $WFSX_p$.
- If $\{\}$ is a support of $\bot$ then the program is unrevisable.
- If there are no supports of $\bot$ then the program is noncontradictory.
- Otherwise, after having all supports of $\bot$, the rest follows by operations on these sets, and computing indissociables. For such operations on sets one can rely on efficient methods known from the literature. For example the method of [Reiter, 1987] for finding minimal diagnosis can be herein applied for finding CRSs given the supports. Example 8.39 shows that the issue of indissociables is simplified when the approch of CRS is considered.
- Finally, a minimal revised program is obtained by adding to P one inhibition rule for each element of a CRS, and the sceptical revision is obtained as the union of all such minimal revised programs.

Example 8.39 Consider program P :

$$\begin{array}{lcl} \bot & \leftarrow & not\ a \\ a & \leftarrow & b \end{array}$$

with $Rev = \{not\ a, not\ b\}$.

The submodels of P are:

$$\begin{array}{lll} \langle \{not\ a, not\ b, \bot\}, & \{\} & \rangle \\ \langle \{not\ b\}, & \{not\ a\} & \rangle \\ \langle \{\}, & \{not\ a, not\ b\} & \rangle \end{array}$$

and thus its only MNS (and the sceptical submodel) is the second one.

Note that there exists no submodel with revision $\{not\ b\}$ because $Ind(\{not\ b\}) = \{not\ a\}$. If such a revision would be considered then the sceptical submodel would be the last one.

The only support of $\bot$ is $\{not\ a\}$, and coincides with the only CRS. Note how the issue of indissociables becomes simplified, since eventhough for submodel it is necessary to compute indissociables in order to find correctly the sceptical submodel, this is not the case for CRSs.

Example 8.40 Recall the *"birds fly"* example from the introduction where

$$Rev = \{not\ abnormal(X)\}.$$

The only support of $\bot$ is:

$$\{not\ abnormal(tweety)\}$$

and so, it coincides with the only CRS.

Thus the only MRP, and sceptical revised program, is the original program augmented with $abnormal(tweety) \leftarrow not\ abnormal(tweety)$, whose *WFSX* is:

$$\{bird(tweety), \neg fly(tweety), not\ fly(tweety), man(socrates)\}$$

as expected.

Example 8.41 Consider the hiking/swimming program (example 8.12):

$$\begin{array}{rcl} hiking & \leftarrow & not\ rain \\ swimming & \leftarrow & not\ rain \\ swimming & \leftarrow & not\ cold\_water \\ \\ \bot & \leftarrow & hiking, swimming \end{array}$$

and let $Rev = \{not\ rain, not\ cold\_water\}$.

The supports of $\bot$ are $\{not\ rain\}$ and $\{not\ rain, not\ cold\_water\}$. Thus its removal sets are:

$$\begin{array}{rcll} \{not\ rain\} \cup \{not\ rain\} & = & \{not\ rain\} & \text{and} \\ \{not\ rain\} \cup \{not\ rain, not\ cold\_water\} & = & \{not\ rain, not\ cold\_water\}. & \end{array}$$

The only CRS is $\{not\ rain\}$, so the only MRP of P, and its sceptical revised program is:

$$\begin{array}{rcl} \bot & \leftarrow & hiking, swimming \\ hiking & \leftarrow & not\ rain \end{array} \qquad \begin{array}{rcl} rain & \leftarrow & not\ rain \\ swimming & \leftarrow & not\ rain \\ swimming & \leftarrow & not\ cold\_water \end{array}$$

whose *WFSX* is:

$$\{not\ cold\_water, swimming\}$$

This results coincides with the WFS_{Opt} calculated in example 8.12.

Example 8.42 Recall the program P of example 8.31:

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ \neg a & & \end{array} \qquad \begin{array}{rcl} b & \leftarrow & not\ c \\ c & & \end{array}$$

with revisables $Rev = \{not\ c\}$, whose paraconsistent $WFSX_p$ is:

$$\{c, \neg a, not\ a, not\ b, a, not\ \neg a\}$$

The supports of $\bot$ result from the union of supports of a and supports of $\neg a$. As the only rule for $\neg a$ is a fact, its only support is $\{\}$. Supports of a are the supports of $not\ b$, and supports of $not\ b$ are the supports of c. Again, as the only rule for c is a fact, its only support is $\{\}$.

Thus the only support of $\bot$ is $\{\}$, and so P is unrevisable.

8.4 Equivalence between avoidance and removal

In this section we discuss the equivalence between the approaches of contradiction avoidance and contradiction removal described in this chapter.

The need for semantics more sceptical than *WFSX* can be seen as showing the inadequacy of the latter for certain problems. The equivalence results show that this is not case since, by

providing a revision process, *WFSX* can deal with the same problems as the more sceptical semantics $WFSX_{Opt}$, and gives the same results.

The advantages of using *WFSX* plus the revision process reside mainly on its simplicity compared to the others, and its properties (studied in section 10.1) that make it amenable for top–down and bottom–up computation procedures. Moreover, the top–down procedures for *WFSX* can be obtained by simple modifications of procedures for WFS [Przymusinski, 1989a, Warren, 1989, Pereira *et al.*, 1991e, Chen and Warren, 1992].

The revision procedure can be implemented as a preprocessor of programs, and the maintenace of noncontradiction might benefit from existing procedures for Truth Maintenance Systems.

In order to prove the main equivalence theorems, we begin by proving two important lemmas. These lemmas state that avoiding a hypothesis in contradiction avoidance is equivalent to adding an inhibition rule for that hypothesis in contradiction removal.

Lemma 8.4.1 *If $P \cup H$ is a complete scenario wrt Opt of a program P with avoidance set S then $P' \cup H$ is a complete scenario of $P' = P \cup IR(S)$.*

Proof: Since the inhibition rules are only added for literals in the avoidance set (thus for literals that do not belong to H) it is clear that $P' \cup H$ is consistent, and every mandatory is in H. It remains to be proven that:

1. if *not* L is acceptable then *not* $L \in H$
2. if *not* $L \in H$ and is not mandatory then it is acceptable

For every hypotheses in S this is ensured because they do not belong to H, and none of them is acceptable once the inhibition rules are added[17].

For hypotheses not in S :

1. If *not* $L \notin S$ is acceptable in $P' \cup H$ then it is also acceptable in $P \cup H$, because the latter, having less rules, provides less evidence to the contrary. It's left to prove that:

   $$\text{if } not\ L \in Acc(P \cup H) \text{ then } not\ L \in Acc_{Opt}(P \cup H).$$

   Assume the contrary, i.e.

   $$not\ L \in Acc(P \cup H) \text{ and } not\ L \notin Acc_{Opt}(P \cup H).$$

   By definiton 8.2.4, this is the case where *not* L is acceptable wrt $P \cup H \cup not\ S$. In this case *not* $L \notin Acc(P' \cup H)$ because, by having the inhibiton rules, some *not* $L' \in S$ provides evidence for L. Thus an hypotheses is contradicted.

2. If *not* $L \notin S$ is not mandatory and is in H then it must belong to $Acc_{Opt}(P \cup H)$, and thus, by definition of acceptable hypothesis wrt *Opt*, it also belongs to $Acc(P \cup H)$. So it can only not belong to $Acc(P' \cup H)$ if some of the inhibition rules provide evidence to L, which can never happen because *not* $L \in Acc_{Opt}(P \cup H)$.

$\diamondsuit$

Lemma 8.4.2 *If $P' \cup H$ is a complete scenario of $P' = P \cup IR(R)$, and $R \subseteq Opt$, then $P \cup H$ is complete wrt Opt.*

Proof: Similar to the proof of lemma 8.4.1. $\diamondsuit$

[17] Note that for any program with the $L \leftarrow not\ L$ rule, *not* L constitutes evidence for L, and thus *not* L can never be acceptable.

Theorem 8.4.3 (Quasi–complete scenaria and MNSs) *$P \cup H$ is a quasi–complete scenario wrt Opt of a program P with an avoidance set S iff $\langle M, S\rangle$ is a MNS of P with revisables Opt, where:*

$$M = \text{WFSX}(P \cup IR(S))$$

Proof:

$\Rightarrow$ By lemma 8.4.1, if $P \cup H$ is a quasi–complete scenario wrt *Opt* of P with an avoidance set S then it is a complete scenario of

$$P' = P \cup \{L \leftarrow not\ L \mid not\ L \in S\}.$$

Moreover, given that $P \cup H$ is a base scenario, by definition of quasi–complete, it is the least complete scenario wrt *Opt* with avoidance set S, and thus the *WFSX* of P'. By definition of quasi–complete no smaller combination of *Opt* exists, i.e. no smaller set of inhibition rules closed under indissociables removes the contradiction. So $\langle M, S\rangle$ is a MNS of P with revisables *Opt*.

$\Leftarrow$ Since $\langle M, S\rangle$ is a MNS of P, M is the least complete scenario of $P \cup IR(S)$. Thus, by lemma 8.4.2, $P \cup H$ is complete wrt *Opt*. Moreover, since S is by definition closed under indissociables, $P \cup H$ is the least complete scenario wrt *Opt* with avoidance set S. Thus it is a base scenario. By definition of MNS, no smaller combination of *Opt* removes the contradiction, and there are no base scenaria with a smaller subset of *Opt*, i.e $P \cup H$ is quasi–complete.

$\diamondsuit$

This theorem states that assuming hypotheses maximally and avoiding the contradiction, corresponds to minimally introducing inhibition rules, and then computing the *WFSX*.

Theorem 8.4.4 (Sceptical revision and WFS_{Opt}) *$P \cup H$ is the WFS_{Opt} of a program P with an avoidance set S iff $\langle M, S\rangle$ is the sceptical submodel of P with revisables Opt.*

Proof: The proof follows directly from theorem 8.4.3 and the fact that the sceptical submodels is the join of MNSs, and WFS_{Opt} is the meet of quasi–complete scenaria. $\diamondsuit$

From these theorem it follows that the rôle of optatives in contradiction avoidance is the same as the rôle of revisables in contradiction removal. Thus the discussion about special criteria to automatically infer optatives from a program, applies directly in the issue of finding special criteria to infer revisables from the program.

8.5 Applications

Finally we show examples of application of contradiction removal to diagnosis, to the debugging of pure Prolog programs, and to reasoning about actions. For further details on the subject the reader is refered to [Pereira *et al.*, 1991g, Pereira *et al.*, 1993d, Pereira *et al.*, 1993e, Pereira *et al.*, 1993c, Pereira *et al.*, 1993b].

8.5.1 Diagnosis

Consider the circuit of figure 8.5 comprised of three OR gates:

The circuit is represented by the extended logic program:

% Correct behaviour of OR gates:

$gate(or, G, 0, 0, 0) \leftarrow not\ ab(G)$ $\quad$ $gate(or, G, 1, 0, 1) \leftarrow not\ ab(G)$

$gate(or, G, 0, 1, 1) \leftarrow not\ ab(G)$ $\quad$ $gate(or, G, 1, 1, 1) \leftarrow not\ ab(G)$

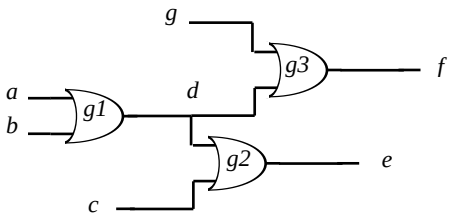

Figure 8.5: OR gates circuit.

% connections among nodes:
$value(d, D) \leftarrow value(a, A), value(b, B), gate(or, g1, A, B, D)$
$value(e, E) \leftarrow value(d, D), value(c, C), gate(or, g2, D, C, E)$
$value(f, F) \leftarrow value(d, D), value(g, G), gate(or, g3, D, G, F)$

% No node can simultaneously have the values 1 and 0:
$\neg value(X, 0) \leftarrow value(X, 1)$
$\neg value(X, 1) \leftarrow value(X, 0)$

To this program we add facts F reporting observations made. Suppose we observe the inputs:

$$\{value(a, 1), value(b, 0), value(c, 0), value(g, 0)\}.$$

$P \cup F$ is noncontradictory, thus no revision is in order, and its semantics is the *WFSX*. In particular:

$$\{not\ ab(g1), not\ ab(g2), not\ ab(g3), value(d, 1), value(f, 1), value(e, 1)\} \subseteq WFSX(P)$$

stating the behaviour of every gate is normal, the values of the nodes are as expected, and all belong to the consequences of *WFSX*.

Suppose an additional factual observation O_1 is made:

$$value(e, 0).$$

Then the program $P \cup F \cup O_1$ is contradictory, and so *WFSX* doesn't assign meaning to it. However revisions exist, where minimal revised programs (PR_1 and PR_2) correspond to minimal diagnoses:

$$\begin{aligned} \{not\ ab(g2), not\ ab(g3), value(e, 0)\} &\subseteq WFSX(PR_1) \\ \{not\ ab(g1), not\ ab(g3), value(d, 1), value(f, 1), value(e, 0)\} &\subseteq WFSX(PR_2) \end{aligned}$$

where $not\ ab(g1) \notin WFSX(PR_1)$ and $not\ ab(g2) \notin WFSX(PR_2)$.

Finally suppose that yet an additional observation O_2 in node d is made, and, its result is:

$$value(d, 0).$$

Then the only minimal revised program PR_3 corresponds to the only possible diagnosis:

$$\{not\ ab(g2), not\ ab(g3), value(d, 0), value(f, 0), value(e, 0)\} \subseteq WFSX(PR_3)$$

8.5.2 Debugging of pure Prolog

In pure Prolog programs, besides looping there are only two other kinds of error [Lloyd, 1987]: wrong solutions or finitely missing solutions. First we apply contradiction removal to perform debugging of wrong solutions of pure Prolog programs, assuming that a Prolog program stands for its ground version. For an elaboration of these techniques the reader is refered to [Pereira *et al.*, 1993c].

Consider the buggy Prolog program P:

$$\begin{array}{rcl} a(1) & & \\ a(X) & \leftarrow & b(X), c(Y,Y) \\ b(2) & & \\ b(3) & & \\ c(1,X) & & \\ c(2,2) & & \end{array}$$

As you can check, goal $a(2)$ succeeds with the above program. Suppose now that $a(2)$ should not be a conclusion of P, so that $a(2)$ is a wrong solution. What are the minimal causes of this bug?

There are three. First, the obvious one, the second rule for a has a bug; the second is $b(2)$ should not hold in P; and finally, that neither $c(1,X)$ nor $c(2,2)$ should hold in P.

This type of error (and its causes) is easily detected using contradiction removal by means of a simple transformation applied to the original program: add

$$not\ incorrect_i(X_1, X_2, \ldots, X_n)$$

to the body of each i-th rule of P, where n is its arity and $X_1, X_2, \ldots, X_n$ its head arguments.

Applying this to P we obtain P_1:

$$\begin{array}{rcl} a(1) & \leftarrow & not\ incorrect_1(1) \\ a(X) & \leftarrow & b(X), c(Y,Y), not\ incorrect_2(X) \\ b(2) & \leftarrow & not\ incorrect_3(2) \\ b(3) & \leftarrow & not\ incorrect_4(2) \\ c(1,X) & \leftarrow & not\ incorrect_5(1,X) \\ c(2,2) & \leftarrow & not\ incorrect_6(2,2) \end{array}$$

Now if we have wrong solution:

$$p(X_1, X_2, \ldots, X_n)$$

in P just add to the extended program P_1 the fact

$$\neg p(X_1, X_2, \ldots, X_n),$$

and to find the possible causes of the wrong solution use as revisables all the $not\ incorrect_i$ hypotheses.

For instance, if $a(2)$ is a wrong solution of program P, by adding $\neg a(2)$ to P_1 we obtain three minimal revisions of P_1 : one undefining $not\ incorrect_1(2)$; another undefining $not\ incorrect_3(2)$; and another undefining both $not\ incorrect_5(1,1)$ and $not\ incorrect_5(2,2)$.

Suppose now a program should not finitely fail on some goal but does so. This is the missing solution problem. Say, for instance, $a(4)$ should succeed in P above. Which are the minimal sets of rules that added to P make $a(4)$ succeed? There are two minimal solutions: either add rule $a(4)$ or rule $b(4)$.

The solution to this type of bug is trickier than the one before, but it suffices to introduce for each predicate p with arity n the following rule:

- $p(X_1, X_2, \ldots, X_n) \leftarrow uncovered(p(X_1, X_2, \ldots, X_n))$

Then all that's necessary is to add the constraint:

$$\bot \leftarrow not\ q(X_1, X_2, \ldots, X_n)$$

to state that if predicate q has a missing solution $q(X_1, X_2, \ldots, X_n)$ then a potential contradiction arises, and obtain the minimal revisions of the transformed program, where revisables are $not\ uncovered(A)$, for all atoms A.

The transformed program P_2 obtained is P plus the rules:

$$\begin{array}{rcl} a(X) & \leftarrow & uncovered(a(X)) \\ b(X) & \leftarrow & uncovered(b(X)) \\ c(X,Y) & \leftarrow & uncovered(c(X,Y)) \end{array}$$

To find the possible causes of the missing solution to $a(4)$ we add the constraint:

$$\bot \leftarrow not\ a(4)$$

and find, as expected, two minimal revisions: one not containing $not\ missing(a(4))$ an d another not containing $not\ missing(b(4))$. This means that by adding $a(4)$ or $b(4)$ to P the missing solution is no longer missed.

8.5.3 Reasoning about actions

Here we study one classical problem of reasoning about actions using the situation calculus, and show how the major drawbacks of other representations can be easily solved by using *WFSX* with the contradiction removal procedures introduced.

Situation calculus has three kinds of entities: fluents, actions and situations. We use the predicate $h(F, S)$ to say that fluent F holds in situation S, and a term $r(A, S)$ represents the new situation obtained as the result of performing action A in situation S. It's also necessary to add a frame axiom which expresses the *"common sense law of inertia"* [McCarthy, 1986] stated in [Lifschitz, 1991a] as:

"In the absence of information to the contrary, properties of objects can be assumed to remain unchanged after an action is performed"

which can be formalized as $[h(F, r(A, S)) \Leftrightarrow h(F, S)] \Leftarrow not\ ab(A, F, S)$ and will be represented by the four rules:

$$\begin{array}{rcl} h(F, r(A, S)) & \leftarrow & h(F, S), not\ ab(A, F, S) \\ \neg h(F, r(A, S)) & \leftarrow & \neg h(F, S), not\ ab(A, F, S) \\ h(F, S) & \leftarrow & h(F, r(A, S)), not\ ab(A, F, S) \\ \neg h(F, S) & \leftarrow & \neg h(F, r(A, S)), not\ ab(A, F, S) \end{array}$$

As [Gelfond and Lifschitz, 1992] explains, the first two rules are used to apply the law of inertia in reasoning from past to future and the other two from future to past.

If the negation of the abnormality predicate is interpreted as classical negation, as in McCarthy's original formulation, it becomes necessary to have the following two extra rules added to the program:

$$\begin{array}{rcl} ab(A, F, S) & \leftarrow & \neg h(F, S), h(F, r(A, S)) \\ ab(A, F, S) & \leftarrow & h(F, S), \neg h(F, r(A, S)) \end{array}$$

But as we shall see, our approach automatically infers the situations that are exceptions to the frame axiom. This is the essence of the frame problem: Having incomplete knowledge about

the world, what properties of objects (fluents) are changed as a result of action A in situation S ?

We discuss a (new) version of the stolen car problem [Kautz, 1986] showing how it is handled using contradiction removal with intuitive results. For sake of simplicity, we do not present other instances of this problem (c.f. [Shanahan, 1992]) that are also correctly handled and easily represented in extended logic programs with contradiction removal.

The formulation of the stolen car problem (SCP) is:

You leave your car parked, return after a while, and your car is gone.
How can you explain that ?

This problem is easily represented in situation calculus. In the initial situation, s_0, the car is parked. After a finite number of wait actions, for instance 4, the car has disappeared. Now suppose that after two wait actions the car was still seen parked by someone. This problem statement is represented by the logic program:

$$\begin{array}{l} h(cp, s_0) \\ h(cp, r(w, r(w, s_0))) \\ \neg h(cp, r(w, r(w, r(w, r(w, s_0))))) \end{array}$$

plus the above four frame axiom rules.

First we must determine what are the supports of contradiction. There are only two:

$$\begin{array}{rcl} SS_1 & = & \{not\ ab(w, cp, s_2), not\ ab(w, cp, s_3)\} \\ SS_2 & = & \{not\ ab(w, cp, s_0), not\ ab(w, cp, s_1), not\ ab(w, cp, s_2), not\ ab(w, cp, s_3)\} \end{array}$$

with $s_i = r(w, s_{i-1}), i \geq 1$. Thus there are two $CRSs$:

$$\begin{array}{rcl} CRS_1 & = & \{not\ ab(w, cp, s_2)\} \\ CRS_2 & = & \{not\ ab(w, cp, s_3)\} \end{array}$$

corresponding to the intuitive result that something abnormal happened during either the third or fourth wait action.

Chapter 9

Adding CWAs to well–founded models

The semantics described in the previous sections, is based on a generalization of the well–founded semantics (WFS) [Gelder *et al.*, 1991] for extended logic programs. However it can be argued [Kakas and Mancarella, 1991b, Pereira *et al.*, 1992a, Pereira *et al.*, 1993a] that sometimes WFS is excessively careful in deciding about the falsity of some atoms, leaving them undefined, and that a suitable form of CWA can be used to safely and undisputably assume false some of the atoms absent in the well–founded model of a program (i.e. undefined ones).

In this chapter we come back to the issue of normal program semantics and, based on a suitable form of CWA, define for such programs how additional negative assumptions are to be added to the WFS of a program. This proffers a novel semantics for normal programs – the O–semantics.

After having defined the O–semantics, we proceed to generalize it to extended logic programs and, finally, compare the resulting semantics with *WFSX*.

Consider the following example adapted from [Kakas and Mancarella, 1991a], itself a variant of the "game" example of [Gelfond and Lifschitz, 1988]:

Example 9.1 The program:

$$\begin{aligned} win(X) &\leftarrow move(X,Y), not\ win(Y) \\ raisedBet(X) &\leftarrow win(X) \\ move(a,a) &\leftarrow \\ move(b,c) &\leftarrow \end{aligned}$$

expresses that:

- "X is a winning position if there is a move from X to Y and Y is not a winning position";
- "in a winning position bets are raised";
- "we can make a move from position a to position a, and from position b to c".

c is not a winning position since it is impossible to move from c. b is a winning position because it is possible to move from b to c and c is not a winning position. a is a draw position.

Neither $win(a)$ nor $not\ win(a)$ should hold. This is correctly handled by WFS, which assigns the truth–value undefined to $win(a)$.

The semantics of this program should also capture the intended meaning that bets are not raised in a position of draw. This is not captured by WFS which leaves $raisedBet(a)$ undefined.

More abstractly, let P :

$$\begin{array}{lcl} c & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

where $WFM(P) = \{\}$. We argue that the intended meaning of the program may be $\{not\ c\}$, since a may not be true in any partial stable model of P via the second rule because that would require the truth of $not\ a$, and so the first rule cannot possibly contradict the assigned meaning. Another way to understand this is that one may safely assume $not\ c$ using a form of CWA on c, since $not\ a$ may not be consistently assumed and so there is no support for c.

However, when relying on the absence of present evidence about some atom A, we do not always want to assume that $not\ A$ holds, since there may exist consistent assumptions allowing to conclude A. Roughly, we want to define the notion of concluding for the truth of a negative literal $not\ A$ just in case there is no hard nor hypothetical evidence to the contrary, i.e. no consistent set of negative assumptions such that $not\ A$ is untenable.

Consider program P :

$$\begin{array}{lcl} c & \leftarrow & a \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

If we interpret the meaning of this program as its WFM (which is empty), and as we do not have a, a naïve CWA could be tempted to derive $not\ c$ based on the assumption $not\ a$. There is however an alternative negative assumption $not\ b$ that, if made, defeats the assumption $not\ a$, i.e. the assumption $not\ a$ may not be sustained since it can be defeated by the assumption $not\ b$. We will define later more precisely the notions of sustainability, defeating, and tenability.

Both the above programs have empty well–founded models. We argue that WFS is overcareful in these cases, and something more can safely be added to the meaning of program, thus reducing its undefinedness, as long as we are willing to adopt a suitable form of CWA.

We contend that a set $CWA(P)$ of negative literals (assumptions) *added* to a program model $MOD(P)$ by CWA must obey the four principles:

1. $MOD(P) \cup CWA(P) \not\models L$ for any $not\ L \in CWA(P)$. This states that the program model added with the set of assumptions identified by the CWA rule must be **consistent**.
2. There is no other set of assumptions A such that $MOD(P) \cup A \models L$ for some $not\ L \in CWA(P)$. I.e. $CWA(P)$ is **sustainable**.
3. $CWA(P)$ must be **unique**.
4. $CWA(P)$ must, additionally, be **maximal**.

9.1 Beyond the WFS of normal programs

In this section we identify the meaning of a normal logic program P as a suitable partial closure of the well–founded model of the program in the sense that it contains the well–founded model (and thus always exists). The extension we propose reduces undefinedness in the intended meaning of a program P, by an adequate form of CWA based on notions of consistency, sustainability and tenability with regard to alternative negative assumptions. Sustainability of a consistent set of negative assumptions insists that there be no other consistent set that defeats it (i.e. there is no hypothetical evidence whose consequences contradict the sustained assumptions). Tenability requires that a maximal sustainable set of assumptions be not contradicted by the consequences of adding to it another competing (nondefeating and nondefeated) maximal sustainable set.

This section is organized as follows: first we introduce adequate definitions for capturing the concepts behind the semantics, accompanied by examples illustrating them. Then models are defined and organized into a lattice, and the class of sustainable A-Models is identified. Next we define the O-Semantics of a normal program P on the basis of the class of maximal sustainable tenable A-Models. A unique model is finally singled out as the O–model of P. Afterwards we present some properties of the class of A–models, and terminate by relating to other semantics.

Parts of this section appear in [Pereira *et al.*, 1992a] and in [Pereira *et al.*, 1993a].

9.1.1 Adding negative assumptions to a program

Here we show how to consistently add more negative assumptions to the well–founded model of a normal program P. Informally, it is consistent to add a negative assumption to $WFS(P)$ if the assumption atom is not among $WFS(P)$ after adding the assumption to P. We also define when a set of negative assumptions is defeated by another, and show how the models of a program, when different sets of negative assumptions added to it, are organized into a lattice.

The addition of assumption to a program is performed differently from the manner of chapter 7. There, although default literals are also treated as assumptions to be added to a program, the addition process starts from scratch. In contrast, here we begin by adding more default literals to the well–founded model of a program, so that we can simplify the process by taking the well–founded semantics and its default literals as given.

So we begin by defining what it means in this context to add new assumptions to the well–founded model of a program. This is achieved by simply substituting *true* for the assumptions to be added, and *false* for their atoms, in all body rules of the program.

Definition 9.1.1 ($P + A$) *The program $P + A$ obtained by adding to a normal program P a set of negative assumptions $A \subseteq not\ \mathcal{H}$ is the result of:*

- *Deleting from P all rules*

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

  such that some $not\ B_i \in A$

- *Deleting from the remaining rules all literals $not\ L \in A$*

Definition 9.1.2 (Assumption model) *An Assumption Model of a program P, or A–model for short, is a pair $\langle A; M\rangle$ where $A \subseteq not\ \mathcal{H}$ and*

$$M = WFM(P + A)$$

The meaning of an A–model $\langle A; WFM(P + A)\rangle$ is:

$$A \cup WFM(P + A)$$

i.e. the result of adding assumptions A to the well–founded model of P.

Among these models we define the partial order $\leq_a$ in the following way:

$$\langle A_1; M_1\rangle \leq_a \langle A_2; M_2\rangle \quad \text{iff} \quad A_1 \subseteq A_2$$

On the basis of set union and set intersection among the sets A of negative assumptions, the set of all A–models becomes organized as a complete lattice.

Having defined assumption models we next consider their consistency. According to the CWA principles above, an assumption *not* A cannot be added to a program P if by doing so A is itself a consequence of P, or if some other assumption is contradicted.

Definition 9.1.3 (Consistent A–model) *An A–model $\langle A; M\rangle$ is consistent iff $A \cup M$ is an interpretation, i.e. there exists no assumption not $L \in A$ such that $L \in M$.*

Example 9.2 Let P :

$$\begin{array}{lcl} c & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ a & \leftarrow & not\ a \end{array}$$

whose WFM is empty.

$P + \{not\ a\}$ is:

$$\begin{array}{lcl} c & \leftarrow & not\ b \\ b & \leftarrow & \\ a & \leftarrow & \end{array}$$

whose WFS is $\{a, b, not\ c\}$. Thus:

$$\langle \{not\ a\}; \{a, b, not\ c\}\rangle$$

is an A–model. This A–model is inconsistent because $a \in WFM(P + \{not\ a\})$. The same happens with all A–models containing the assumption *not* a. The A–model:

$$\langle \{not\ b, not\ c\}; \{c\}\rangle$$

is also inconsistent.

Thus the only consistent A–models are

$$\begin{array}{l} \langle \quad \{\} \quad ; \ \{\} \ \rangle \\ \langle \ \{not\ b\} \ ; \ \{c\} \ \rangle \\ \langle \ \{not\ c\} \ ; \ \{\} \ \rangle \end{array}$$

In this example we see that $\langle \{not\ a\}; \{a, b, not\ c\}\rangle$ is inconsistent and the same happens for all A–models containing the assumption *not* a. Indeed, and in general:

Lemma 9.1.1 *If an A–model AM is inconsistent then any A–model AM' such that $AM \leq_a AM'$ is inconsistent.*

Proof: In appendix. $\diamondsuit$

According to the CWA principles above, an assumption *not* A cannot be sustained if there is some set of consistent assumptions that concludes A. We've already expressed the notion of consistency being used. To capture the notion of sustainability we now formally define how an A–model can defeat another, and define sustainable A–models as the non–defeated consistent ones.

Definition 9.1.4 (Defeating) *A consistent A–model $\langle A; M\rangle$ is defeated by the consistent A–models $\langle A'; M'\rangle$ iff*

$$\exists not\ a \in A | a \in M'.$$

Definition 9.1.5 (Sustainable A–models) *An A–model $\langle A; M\rangle$ is sustainable iff it is consistent and not defeated by any consistent A–model, i.e. $\langle not\ S; M\rangle$ is sustainable iff:*

$$S \cap \bigcup_{consistent\ \langle A_i; M_i\rangle} M_i = \{\}$$

Example 9.3 The only sustainable models in example 9.2 are:

$$\begin{array}{l} \langle \quad \{\} \quad ; \{\} \ \rangle \\ \langle \ \{not\ b\} \ ; \{c\} \ \rangle \end{array}$$

Note that the consistent A–model $\langle\{not\ c\};\{\}\rangle$ is defeated by $\langle\{not\ b\};\{c\}\rangle$, i.e. the assumption *not c* is unsustainable since there is a set of consistent assumptions (namely $\{not\ b\}$) that leads to the conclusion c.

The assumptions part of maximal sustainable A–models of a program P are maximal sets of consistent Closed World Assumptions that can be safely added to the well–founded model of P without risking inconsistency by the making of other assumptions.

Lemma 9.1.2 *If an A–model AM is defeated by another A–model D, then all A–models AM′ such that $AM \leq_a AM'$ are defeated by D.*

Proof: If $AM = \langle A; M\rangle$ is defeated by $D = \langle A_D; M_D\rangle$, then there exists $d \in M_D$ such that $not\ d \in A$. Since all AM's are of the form $AM' = \langle A'; M'\rangle$ where $A' = A \cup B$ then $not\ d \in A'$, i.e. D defeats AM'. $\diamondsuit$

Lemma 9.1.3 *The A–model $\langle\{\}; WFM(P)\rangle$ is always sustainable.*

Proof: By definition of sustainable. $\diamondsuit$

Theorem 9.1.4 *The set of all sustainable A–models of a program is nonempty. On the basis of set union and set intersection among their A sets, the A–models ordered by $\leq_a$ form a lower semilattice.*

Proof: Follows directly from the above lemmas. $\diamondsuit$

A program may have several maximal sustainable A–models.

Example 9.4 Let P be:

$$\begin{array}{lcl} c & \leftarrow & not\ c, not\ b \\ b & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

Its sustainable A–models are:

$$\begin{array}{l} \langle \quad \{\} \quad ; \{\} \ \rangle \\ \langle \ \{not\ b\} \ ; \{\} \ \rangle \\ \langle \ \{not\ c\} \ ; \{\} \ \rangle \end{array}$$

The last two are maximal sustainable A–models. We cannot add both *not b* and *not c* to the program to obtain a sustainable A–model since

$$\langle\{not\ b, not\ c\};\{c\}\rangle$$

is inconsistent.

9.1.2 The O–semantics

This subsection is concerned with the problem of singling out, among all sustainable A–models of a program P, one that uniquely determines the meaning of P when the CWA is enforced. This is accomplished by means of a selection criterium that takes a lower semilattice of sustainable A–models and obtains a subsemilattice of it, by deleting A–models that in a well–defined sense are less preferable, i.e. the untenable ones.

Sustainability of a consistent set of negative assumptions insists that there be no other consistent set that defeats it (i.e. there be no hypothetical evidence whose consequences contradict the sustained assumptions). Tenability requires that a maximal sustainable set of assumptions be not contradicted by the consequences of adding to it another competing (nondefeating and nondefeated) maximal sustainable set.

The selection process is repeated and ends up with a complete lattice of sustainable A–models which for every program P is by definition its O–semantics. The meaning of P is then specified by the greatest A–model of the semantics, its O–Model.

To illustrate the problem of preference among maximal A–models recall example 9.4 above. Because we wish to maximize the number of negative assumptions we consider the maximal A–models, which in that case are:

$$\langle\ \{not\ b\}\ ;\ \{\}\ \rangle$$
$$\langle\ \{not\ c\}\ ;\ \{\}\ \rangle$$

The join of these maximal A–models:

$$\langle \{not\ b, not\ c\}; \{c\} \rangle$$

is per force inconsistent, in this case wrt c. This signifies that when assuming *not* c there is an additional set of assumptions entailing c, making this A–model *untenable*. But the same does not apply to *not* b. So we can say that *not* b is more primal than *not* c, in the sense that when added with *not* c it causes an inconsistency in c but does not affect itself so. In the program that is indeed intuitively the case, i.e. *not* c depends on *not* b but not vice–versa, and so *not* b is more primal[1].

Thus the preferred A–model is

$$\langle \{not\ b\}, \{\} \rangle$$

and the A–model $\langle \{not\ c\}; \{\} \rangle$ is said untenable.

The rationale for the preference is grounded in that the inconsistency of the join arises wrt c but not wrt b.

Definition 9.1.6 (Candidate structure) *A Candidate Structure CS of a program P is any subsemilattice of the lower semilattice of all sustainable A–models of P.*

Definition 9.1.7 (Untenable A–models) *Let $\{\langle A_k; M_k \rangle \mid k \in K\}$ for $k \in K$, be the set of all maximal A–models in Candidate Structure CS, and let*

$$J = \langle A_J; M_J \rangle$$

be the join of all such A–models, in the complete lattice of all A–models.

An A–model $\langle A_i; M_i \rangle$ is untenable wrt CS iff it is maximal in CS and there exists not $a \in A_i$ such that $a \in M_J$.

[1]Note the complementarity between this notion of primacy and the one of primacy of hypotheses described in section 8.2.1. There, *not* A is more primal than *not* B if by adding *not* A, *not* B follows as a consequence. Here, *not* A is more primal than *not* B if, by adding both *not* A and *not* B, B follows as a consequence while A doesn't.

Proposition 9.1.1 *There exists no untenable A–model wrt a Candidate Structure with a single maximal element.*

Proof: Since the join coincides with the unique maximal A–model, which is sustainable by definition of CS, then it cannot be untenable. ♢

The candidate structure left after removing all untenable A–models of a CS may itself have several untenable elements, some of which might not be untenable A–models in the initial CS. If the removal of untenable A–models is performed repeatedly on the retained Candidate Structure, a structure with no untenable models is eventually obtained, albeit the bottom element of the candidate structure.

Definition 9.1.8 (Retained CS) *The Retained Candidate Structure $R(CS)$ of a Candidate Structure CS is defined recursively in the following way (where J is the join of elements of CS in the complete lattice of all A–models):*

- *$J \cup CS$ if there are no untenable A–models in CS.*
- *Otherwise, let Unt be the set of all untenable A–models wrt CS. Then $R(CS) = R(CS - Unt)$*

Definition 9.1.9 (The O–semantics and the O–model) *The O–semantics of a program P is defined by the Retained Candidate Structure of the semilattice of all sustainable A–models of P.*

Let $\langle A; M\rangle$ be its maximal element. The intended meaning of P is $A \cup M$, the O–Model of P.

Remark 9.1.1 *At this point, we are in a position to make an important remark. Our goal is to maximally reduce undefinedness of the well–founded model by adding to it negative assumptions. Now, the peeling process of subtracting only* maximal *untenable A–models from candidate structures ends up with a retained candidate structure with a maximal element. So we must guarantee this element is always greater or equal than the result we would obtain if we didn't require untenable A–models to be* maximal *in definition 9.1.7.*

This is indeed guaranteed, for the join of the maximal elements of each candidate structure is always greater than any join of non–maximal elements of that structure, and because the maximal element of the retained lattice is by definition one such join of maximal elements.

Example 9.9 shows that if untenable A–models were not defined as maximal then a smaller O–Model would be obtained.

Theorem 9.1.5 *The O–semantics of a program P is always defined by a complete lattice of sustainable A–models.*

Proof: Since every candidate structure is a semilattice of sustainable A–models, it is enough to prove that the join $J = \langle A_J; M_J\rangle$ of the retained candidate structure CS of the semilattice of all sustainable A–models of P is a sustainable A–model.

If we assume that J is inconsistent then at least one maximal A–model in CS is untenable. Accordingly, since in the final retained CS there are, by definition, no untenable A–models, J is consistent.

J cannot be defeated by any other consistent A–model D because, in such a case, at least one other element of CS would also be defeated by D, which is impossible by definition of candidate structure. ♢

Corollary 9.1.1 *The O–semantics of a program has no untenable A–model wrt itself.*

Proof: Follows directly from the theorem and proposition 9.1.1. ♢

Corollary 9.1.2 (Existence of the O–semantics) *The Retained Candidate Structure of the semilattice of all sustainable A–models is nonempty.*

Proof: Follows directly from the theorem. ♢

9.1.3 Examples

In this subsection we display some examples and their O–semantics. Remark that indeed the O–Models obtained express the safe CWAs compatible with the WFMs (which are all {}). In subsection 9.1.5, "Relation to other work" additional examples can be found which bring out the distinctness of O–semantics wrt other semantics.

Example 9.5 Let P be:

$$\begin{array}{lcl} d & \leftarrow & c \\ c & \leftarrow & not\ c, not\ b \\ b & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

The semilattice of all sustainable A–models CS is depicted in figure 9.1, where shadowed A–models are the tenable ones.

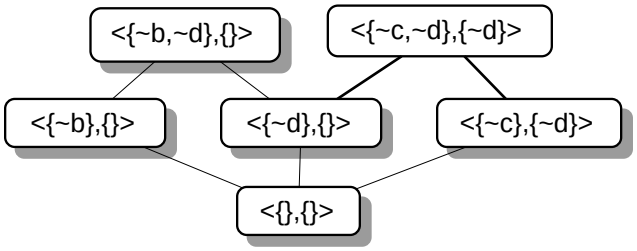

Figure 9.1: Semilattice of all sustainable A–models of example 9.5.

The join of its maximal A–models is:

$$\langle \{not\ b, not\ c, not\ d\}; \{c, not\ d\} \rangle$$

Consequently, the maximal A–model on the right is untenable since it contains $not\ c$ in the assumptions, and c is a consequence of the join.

So $R(CS) = R(CS')$ where CS' is is the candidate structure of figure 9.2.

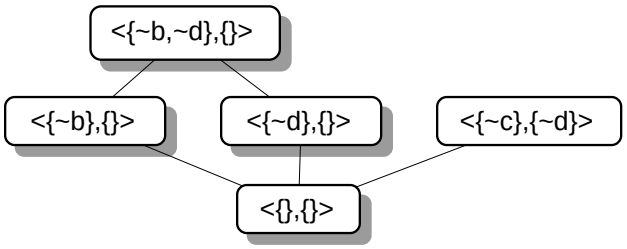

Figure 9.2: Retained candidate structure CS' of example 9.5.

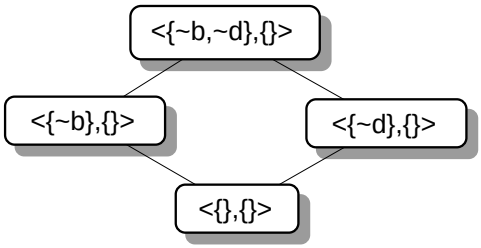

Figure 9.3: O–semantics CS'' of example 9.5.

The join of all maximal elements in CS' is the same as before and the only untenable A–model is again the maximal one having *not c* in its assumptions. Thus $R(CS) = R(CS'')$ where CS'' is depicted in figure 9.3.

So the O–Model is $\{not\ b, not\ d\}$.

Note that if P is divided into P_1 :

$$\begin{array}{lcl} d & \leftarrow & c \\ c & \leftarrow & not\ c, not\ b \end{array}$$

and P_2 :

$$\begin{array}{lcl} b & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

the O–models of P_1 and P_2 both agree on the only common literal *not b*. So *not b* rightly belongs to the O–model of P.

Example 9.6 Consider P :

$$\begin{array}{lcl} q & \leftarrow & not\ p \\ p & \leftarrow & a \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ c \\ c & \leftarrow & not\ a \end{array}$$

Its only consistent A–models are:

$$\begin{array}{l} \langle \quad \{\} \quad ; \{\} \ \rangle \\ \langle \ \{not\ p\} \ ; \{q\} \ \rangle \\ \langle \ \{not\ q\} \ ; \ \{\} \ \rangle \end{array}$$

As this last one is defeated by the second, the only sustainable ones are the first two. Since only one is maximal, these two A–models determine the O–semantics, and the meaning of P is $\{not\ p, q\}$, its O–Model.

Note that if the three last rules, forming an *"undefined loop"*, are replaced by another *"undefined loop"* $a \leftarrow not\ a$, the O–model is the same. This is as it should, since the first two rules conclude nothing about a.

Example 9.7 Let P :

$$\begin{array}{lcl} p & \leftarrow & a,\ b \\ a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

The A–models with *not b* in their assumptions defeat A–models with *not a* in their assumptions and vice-versa. Thus the O–semantics is determined by the A–models:

$$\begin{array}{l} \langle \quad \{\} \quad ; \{\} \ \rangle \\ \langle \ \{not\ p\} \ ; \{\} \ \rangle \end{array}$$

and the meaning of P is $\{not\ p\}$, its O–Model.

Example 9.8 Consider the program P :

$$\begin{array}{lcl} c & \leftarrow & not\ c,\ not\ b \\ b & \leftarrow & not\ c,\ not\ b \\ b & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

Its sustainable A–models are:

$$\begin{array}{l} \langle\ \ \{\}\ \ ;\{\}\ \rangle \\ \langle\ \{not\ b\}\ ;\{\}\ \rangle \\ \langle\ \{not\ c\}\ ;\{\}\ \rangle \end{array}$$

The join of the two maximal ones is

$$\langle \{not\ b, not\ c\}; \{b, c\} \rangle$$

and so both are untenable. Thus the retained candidate structure has the single element $\langle \{\}; \{\} \rangle$ and the meaning of P is $\{\}$.

Example 9.9 Consider the program P :

$$\begin{array}{lcl} c & \leftarrow & not\ a,\ not\ c \\ c & \leftarrow & not\ b,\ not\ c \\ a & \leftarrow & not\ b,\ not\ c \\ a & \leftarrow & d \\ b & \leftarrow & d \\ c & \leftarrow & d \\ d & \leftarrow & not\ d \end{array}$$

The semilattice of all sustainable A–models CS is presented in figure 9.4.

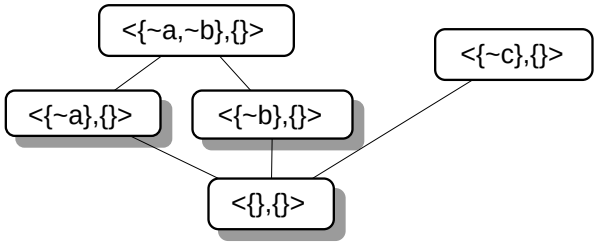

Figure 9.4: Semilattice of all sustainable A–models of example 9.9.

The join of its maximal A–models is

$$\langle \{not\ a, not\ b, not\ c\}; \{a, c\} \rangle$$

Consequently, all maximal A–models are untenable. So $R(CS) = R(CS')$ where CS' is the structure in figure 9.5.

Since the join of all elements is

$$\langle \{not\ a, not\ b\}; \{\} \rangle$$

there are no untenables in CS'. The O–semantics is as shown in figure 9.6, and the O–Model is $\{not\ a, not\ b\}$.

If untenable A–models were not defined as maximal ones (cf. remark 9.1.1) then

$$\langle \{not\ a\}; \{\} \rangle$$

would also be untenable wrt to the semilattice of all sustainable A–models CS. Then the $R(CS)$ would be as in figure 9.7, and the O–Model would be smaller: $\{not\ b\}$.

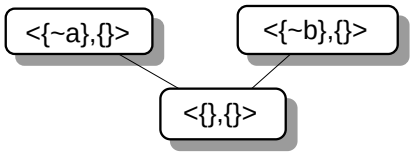

Figure 9.5: Retained candidate structure CS' of example 9.9.

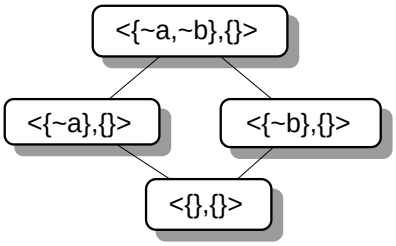

Figure 9.6: O–semantics of example 9.9.

9.1.4 Properties of sustainable A–models

This subsection explores properties of sustainable A–models that provide a better understanding of them, and also give hints for their construction without having to previously calculate all A–models.

We begin with properties that show how our models can be viewed as an extension to well–founded semantics. As mentioned in [Monteiro, 1992], negation in WFS is based on the notion of support, i.e. a literal *not* L only belongs to a partial stable model (PSM) if all the rules for L (if any) have false bodies in the PSM. In contradistinction, we are interested in negations as consistent hypotheses that cannot be defeated. To that end we weaken the necessary (but not sufficient) conditions for a negative literal to belong to a model as explained below. We still want to keep the necessary and sufficient conditions of support for positive literals. More precisely, knowing that PSMs must obey, among others, the following conditions cf. [Monteiro, 1992]:

- If there exists a rule $p \leftarrow Body$ in the program such that $Body$ is true in model M then p is also true in M (*sufficiency of support for positive literals*).
- If an atom $p \in M$ then there exists a rule $p \leftarrow Body$ in the program such that $Body$ is true in M (*necessity of support for positive literals*).
- If all rule bodies for p are false in M then *not* $p \in M$ (*sufficiency of support for negative literals*).
- If *not* $p \in M$ then all rules for p have false bodies in M (*necessity of support for negative literals*).

The meaning of our consistent A–models need not obey the fourth condition. Foregoing it condones making additional negative assumptions. In our models an atom might be false even if it has a rule whose body is undefined. Thus, only false atoms with an undefined rule body are candidates for having their negation added to the $WFM(P)$.

Proposition 9.1.2 *Let $\langle A; M\rangle$ be any consistent A–model of a program P. The interpretation $A \cup M$ obeys the first three conditions above.*

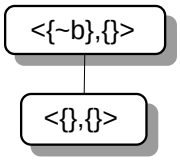

Figure 9.7: $R(CS)$ if untenable A–models were not defined as maximal ones.

Proof: Here we prove the satisfaction of the first condition. The remaining proofs are along the same lines.

If:

$$\exists p \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \in P \mid \{b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m\} \subseteq A \cup M$$

then

$$b_i \in M\ (1 \leq i \leq n)\ and\ not\ c_j \in M\ or\ not\ c_j \in A\ (1 \leq j \leq m).$$

Let

$$p \leftarrow b_1, \ldots, b_n, not\ c_l, \ldots, not\ c_k \quad (l \geq 1, k \leq m)$$

be the rule obtained from an existing one by removing all $not\ c_j \in A$, which is, by definition, a rule of $P + A$. Thus there exists a rule

$$p \leftarrow Body$$

in $P + A$ such that

$$Body \subseteq WFM(P + A) = M.$$

Given that the WFM of any program must obey the first condition above, $p \in WFM(P + A)$. $\diamondsuit$

Next we state properties useful for more directly finding the sustainable A–models.

Proposition 9.1.3 *There exists no consistent A–model $\langle A; M\rangle$ of P with $\{not\ a\} \subseteq A$ such that $a \in WFM(P)$.*

Proof: We begin by proving the proposition for $\{not\ a\} = A$.

Since $a \in WFM(P)$, then by propositions B.0.1 and B.0.2 there is a $SS_P(a) = S$ (according to the definition B.0.1 of support set for normal programs) such that $a \notin S$ and $not\ a \notin S$, and consequently:

$$Rules(S) \subseteq P + \{not\ a\}.$$

Then, by proposition B.0.3, $a \in WFM(P + \{not\ a\})$, and thus

$$\langle \{not\ a\}; WFM(P + \{not\ a\})\rangle$$

is inconsistent.

It follows, from lemma 9.1.1, that all A–models $\langle A; M\rangle$ such that $\{not\ a\} \subseteq A$ are inconsistent. $\diamondsuit$

Hence, A–models not obeying the above restriction are not worth considering as sustainable.

Proposition 9.1.4 *If a negative literal not $L \in WFM(P)$ then there is no consistent A–model $\langle A; M\rangle$ of P such that $L \in M$.*

Proof: We prove that if $L \in M$ for a given A–model $\langle A; M\rangle$ of P then $\langle A; M\rangle$ is inconsistent.

If $L \in M$ there must exist in P a rule

$$L \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

such that:

$$\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\} \subseteq M \cup A$$

and $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ is false in $WFM(P)$[2], i.e. there must exist a rule with head L in P with at least one body literal true in $M \cup A$ and false in $WFM(P)$.

If that literal is some $not\ C_j$, by proposition 9.1.3 $\langle A; M\rangle$ is inconsistent (its corresponding atom is true in $WFM(P)$ and false in $M \cup A$).

If it is some B_i this theorem applies recursively, ending up in a rule with empty body, an atom with no rules or a loop without an interposing *not l*. The truth value of literals in these conditions can never be changed: since the $P + A$ operation only involves deleting rules with literals at the body and literals from the body of rules, the truth value of atoms without rules is always false no matter which A is being considered, and the truth value of atoms with a fact is always false. Literals in a loop without interposing *not l* are false in P, and remain false if rules of the loop are deleted. ♢

Theorem 9.1.6 *If not $L \in WFM(P)$ then for every consistent A–model $\langle A; M\rangle$ of P, not $L \in M$.*

Proof: Given proposition 9.1.4, it suffices to prove that L is not undefined in any consistent A–model of P. The proof is along the lines of that of the proposition above. ♢

Consequently, all negative literals in the $WFM(P)$ belong to every sustainable A–model.

Lemma 9.1.7 *Let $WFM(P) = T \cup not\ F$. For any subset S of not F*

$$WFM(P) = WFM(P + S).$$

Proof: This lemma is easily shown using the definition of $P+A$ and the properties of the WFM. ♢

Theorem 9.1.8 *Let $WFM(P) = T \cup not\ F$, $\langle A; WFM(P + A)\rangle$ a consistent A–model, and let $A' = A \cap not\ F$. Then:*

$$WFM(P + A) = WFM(P + (A - A')).$$

Proof: Let $P' = P + (A - A')$ and $WFM(P) = T \cup not\ F$.

By theorem 9.1.6 $not\ F \subseteq WFM(P')$. So, by lemma 9.1.7:

$$WFM(P') = WFM(P' + not\ F) = WFM([P + (A - (A \cap not\ F))] + not\ F).$$

By definition of $P + A$, it follows that

$$(P + A_1) + A_2 = P + (A_1 \cup A_2).$$

Thus $WFM(P')$ is:

$$WFM(P + [(A - (A \cap not\ F)) \cup not\ F]) = WFM(P + A)$$

[2] $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ is false in a model $M = T \cup not\ F$ iff $\{B_1, \ldots, B_n\} \cap F \neq \{\}$ or $\{C_1, \ldots, C_m\} \cap T \neq \{\}$

$\diamondsuit$

N.B. This theorem shows that sets of assumptions including negative literals of $WFM(P)$ are not worth considering since there exist smaller sets having exactly the same consequences $A \cup M$ and, by proposition 9.1.4, the larger sets are not defeatable by reason of negative literals from the $WFM(P)$. Hence, in the remainder of the chapter, we consider only A–models whose assumptions are not in the WFM, inasmuch all $WFM(P)$ assumptions will be part of $WFM(P + A)$ for any A.

Another important hint for calculating the sustainable A–models is given by lemma 9.1.1. According to it one should start by calculating A–models with smaller assumption sets, so that when an inconsistent A–model is found, by the lemma 9.1.1, sets of assumptions containing it are unworthy of consideration.

Example 9.10 Let P :

$$\begin{array}{lcl} p & \leftarrow & not\ a, not\ b \\ a & \leftarrow & c, d \\ c & \leftarrow & not\ c \\ d & & \end{array}$$

The least A–model is:

$$\langle \{\}; \{d, not\ b\} \rangle$$

where $\{d, not\ b\} = WFM(P)$. Thus sets of assumptions containing $not\ d$ or $not\ b$ are not worth considering.

Take now, for example, the consistent A–model

$$\langle \{not\ a\}; \{d, not\ b, p\} \rangle$$

which we retain since it is consistent.

Consider $\langle \{not\ c\}; \{c, a, not\ p\} \rangle$; as this A–model is inconsistent we do not retain it nor consider any other A–models with assumption sets containing $not\ c$.

Now we are left with just two more A–models worth considering:

- $\langle \{not\ p\}; \{d, not\ b\} \rangle$, which is defeated by $\langle \{not\ a\}; \{d, not\ b, p\} \rangle$.
- $\langle \{not\ p, not\ a\}; \{d, not\ b, p\} \rangle$, which is inconsistent.

Thus the only two sustainable A–models are:

$$\begin{array}{l} \langle \quad \{\} \quad ; \ \{d, not\ b\} \ \rangle \\ \langle \ \{not\ a\} \ ; \ \{d, not\ b, p\} \ \rangle \end{array}$$

In this case, the latter is the single maximal sustainable A–model, and thus uniquely determines the intended meaning of P to be:

$$A \cup WFS(P + A) = \{not\ a, d, not\ b, p\}.$$

9.1.5 Relation to other work

Consider the following program P :

$$\begin{array}{lcl} p & \leftarrow & q, not\ r, not\ s \\ q & \leftarrow & r, not\ p \\ r & \leftarrow & p, not\ q \\ s & \leftarrow & not\ p, not\ q, not\ r \end{array}$$

In [Przymusinska and Przymusinski, 1990] the authors argue that the intended semantics of this program should be the interpretation:

$$\{s, not\ p, not\ q, not\ r\}$$

due to the mutual circularity of p, q, r.

This model is precisely the meaning assigned to the program by the O–semantics, its O–Model. Note that WFS identifies the empty model as the meaning of the program. This is also the model provided by stable model semantics [Gelfond and Lifschitz, 1988]. The weakly perfect model semantics for this program is undefined as noticed in [Przymusinska and Przymusinski, 1990].

The extended well-founded semantics (EWFS) [Dix, 1991] is also an extension to the WFS based on the notion of generalized closed world assumption (GCWA) [Minker, 1987]. Roughly, EWFM moves closer than the WFM (in the sense of being less undefined) to being the intersection of all minimal Herbrand models of P. With a different notation from that of [Dix, 1991]:

$$EWFM(P) =_{def} WFM(P) \cup T(WFM(P)) \cup not\ F(WFM(P))$$

where:

$$T(\mathcal{I}) =_{def} True(MIN\_MOD(\mathcal{I}, P)),$$

$$F(\mathcal{I}) =_{def} False(MIN\_MOD(\mathcal{I}, P))$$

$\mathcal{I}$ is a three–valued interpretation, and $MIN\_MOD(\mathcal{I}, P)$ is the collection of all minimal two–valued Herbrand models of P consistent with $\mathcal{I}$. For a set $\mathcal{S}$ of interpretations, $True(\mathcal{S})$ (resp. $False(\mathcal{S})$) denotes the set of all atoms which are true (resp. false) in all interpretations of $\mathcal{S}$.

For the program $P = \{a \leftarrow not\ a\}$ we have:

$$WFM(P) = \{\}, MIN\_MOD(\{\}, P) = \{\{a\}\} \quad EWFM(P) = \{a\}$$

The O–Model of P is empty.

The main differences between ours and their approach are that:

- Like WFS and unlike EWFM, we insist on the supportedness of positive literals, i.e.:

$$\text{An atom } A \in M_P \text{ iff } \exists\ A \leftarrow Body\ \mid\ Body \subseteq M_P$$

- Unlike WFS and unlike EWFM, we relax, by allowing undefined bodies with false heads under certain conditions, the requirement of supportedness of negative literals, i.e. we relax:

$$not\ A \in M_P \text{ iff } \forall\ A \leftarrow Body\ \mid\ Body \text{ is false in } M_P$$

Example 9.11 Let P be:

$$\begin{array}{lcl} c & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ a & \leftarrow & not\ a \end{array}$$

The O–Model of P is $\{c, not\ b\}$. Note that c has a rule whose body is true in the O–Model and it is not the case that all rules for b are false in it.

The $EWFM$ is $\{a, c, not\ b\}$. The atom a is true in the $EWFM$ and has no rule with a body true in it. All rules for b have a false body in the $EWFM$.

Another example where O–semantics differs from EWFM is the game example of the introduction. In this example EWFM gives the (strange) result that a is a winning position, and thus bets are raised.

A similar approach based on the notion of stable negative hypotheses (built upon the notion of consistency) is introduced in [Kakas and Mancarella, 1991b], identifying a stable theory associated with a program P as the semantics for P, and always contains the well–founded model.

One example showing that their approach is still conservative is:

$$\begin{array}{lcl} p & \leftarrow & not\ q \\ q & \leftarrow & not\ r \\ r & \leftarrow & not\ p \\ s & \leftarrow & p \end{array}$$

Stable theories identify the empty set as the meaning of the program; however its O–Model is $\{not\ s\}$, since it is consistent, maximal, sustainable and tenable.

Kakas and Mancarella (personal communication) now also obtain this model, as a result of the investigation mentioned in the conclusions of [Kakas and Mancarella, 1991b]. In this recent work, the "acceptability semantics", instead of our notion of "sustainable" they present:

A is **KM–coherent**[3] if all sets of assumptions B that defeat A are defeated by A.

i.e., if one insists on the set of assumptions A, no consistent evidence to the contrary can be found. No preferred unique model is identified in their approach, the semantics being defined as the set of consequences common to all KM–coherent sets of assumptions.

However, even with this definition their approach is still conservative (as noted in [Bondarenko *et al.*, 1993]). For example consider the program:

$$\begin{array}{lcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a, not\ b \end{array}$$

Its consistent A–models are:

$$\begin{array}{l} \langle \quad \{\} \quad ; \quad \{\} \quad \rangle \\ \langle\ \{not\ a\}\ ; \quad \{\} \quad \rangle \\ \langle\ \{not\ b\}\ ;\ \{a, not\ b\}\ \rangle \end{array}$$

Given that the second A–model is defeated by the third one, the O–model is $\{a, not\ b\}$, i.e. $not\ b$ is added.

Using the acceptability semantics approach, both $\{not\ a\}$ and $\{not\ b\}$ are KM–coherent, and so the semantics is $\{\}$.

More recently, in [Bondarenko *et al.*, 1993], a flexible framework for defining logic programming semantics is presented, based on an argumentation framework similar to the one in [Dung, 1991] but where a different notion of evidence to the contrary, and of how a scenario defeats contrary evidence, are used.

Within this framework the authors can define the well–founded, stable models, preferred extensions, and acceptability semantics. Then, motivated by the very same example present here that shows the acceptability semantics is too conservative, they define what they call an "improved semantics".

[3]The authors just call it coherent. Here we use KM–coherent to avoid confusion with our coherence principle, which is completely unrelated.

This "improved semantics", however differently defined, is technically equivalent to the preferred extensions semantics of [Dung, 1991] (which for normal programs is equivalent to definition 7.5.1), but where now inconsistent evidence to the contrary is not allowed.

However, the "improved semantics" exhibits some strange behaviours:

Example 9.12 Consider a program with the single rule:

$$a \leftarrow not\ a, not\ b$$

whose WFS and O–model is $\{not\ b\}$. Note that this is indeed the expected result: since b has no rules, $not\ b$ must belong to the semantics independently from the truth value of a. Thus for the standpoint of $not\ a$ this program should be equivalent to the one containing the single rule $a \leftarrow not\ a$, and so $not\ a$ cannot be safely added.

However according to the "improved semantics" $\{not\ a\}$ is a preferred extension of this program, since $P \cup \{not\ a\}$ is consistent and the only evidence for a, $\{not\ a, not\ b\}$, is inconsistent.

9.2 O–semantics for extended programs

Having defined how to add more CWA assumptions to the well–founded model of a normal program, we come back to the issue of semantics of extended program. In this section we generalize O–semantics to extended logic programs.

To that end, we retain the desired principles for adding CWA assumptions in normal programs, i.e the set of added assumptions must be consistent, sustainable, unique and maximal. Here our aim is to add assumptions to the *WFSX* of extended programs, in the same spirit as we did above for the WFS of normal programs. Since for some programs *WFSX* is not defined, for them O–semantics is not defined either. However we want O–semantics to give a meaning to at least the same programs as *WFSX* does.

In order to achieve such a generalization we follow the definition of the O–semantics for normal programs, and modify it appropriately in order to make it applicable to extended programs. In the presentation we also point out whether each of the properties verified in normal programs subsists in extended ones. For those that do not subsist we present alternative (weak) properties that are still verified. This way we obtain some properties of O–semantics for extended programs.

The $P + A$ program transformation for adding assumptions to a program (definition 9.1.1) remains the same for extended programs. In fact all we have to do is to substitute *true* for the assumption, and *false* for their objective literals, in the body of rules. Because making an assumption $not\ L$ does not directly interfere with objective literals, nothing more needs to be done in this stage[4].

The generalization of A–models is the straightforward one, i.e. instead of adding the extra assumptions to the WFS of a program, one adds them to its *WFSX*. Care must be taken bacause, unlike the case of WFS for normal programs, now not every program has a semantics.

Definition 9.2.1 (Assumption model) *An assumption model of an extended program P, or A–model for short, is a pair $\langle A; M\rangle$ where $A \subseteq not\ \mathcal{H}$, $P + A$ is a noncontradictory program, and*

$$M = \mathrm{WFSX}(P + A)$$

[4]Note that adding an objective literal L directly interferes with default literals via coherence ($not\ \neg L$ is a direct consequence of adding L). The converse does not hold.

The meaning of an A–model $\langle A; \mathrm{WFSX}(P + A)\rangle$ *is:*

$$A \cup \mathrm{WFSX}(P + A)$$

i.e. the result of adding assumptions A *to the* WFSX *of* P.

Unlike the case of normal programs here, because of contradiction, A–models might not exist for some sets of assumptions. Thus, in general A–models with the partial order $\leq_a$ do not become organized into a complete lattice.

Example 9.13 Consider program P :

$$\begin{array}{rcl} \neg a & \leftarrow & not\ b \\ a & \leftarrow & not\ c \end{array} \qquad \begin{array}{rcl} b & \leftarrow & not\ d \\ c & \leftarrow & not\ d \end{array} \qquad \begin{array}{rcl} d & \leftarrow & not\ d \end{array}$$

The A–models of P are:

$$\begin{array}{rcl}
M_1 & = & \langle \quad \{\} \quad ; \quad \{not\ \neg b, not\ \neg c, not\ \neg d\} \quad \rangle \\
M_2 & = & \langle \quad \{not\ b\} \quad ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_3 & = & \langle \quad \{not\ c\} \quad ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_4 & = & \langle \quad \{not\ a\} \quad ; \quad \{not\ \neg b, not\ \neg c, not\ \neg d\} \quad \rangle \\
M_5 & = & \langle \quad \{not\ \neg a\} \quad ; \quad \{not\ \neg b, not\ \neg c, not\ \neg d\} \quad \rangle \\
M_6 & = & \langle \quad \{not\ a, not\ \neg a\} \quad ; \quad \{not\ \neg b, not\ \neg c, not\ \neg d\} \quad \rangle \\
M_7 & = & \langle \quad \{not\ a, not\ b\} \quad ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_8 & = & \langle \quad \{not\ \neg a, not\ b\} \quad ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_9 & = & \langle \{not\ a, not\ \neg a, not\ b\} ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_{10} & = & \langle \quad \{not\ a, not\ c\} \quad ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_{11} & = & \langle \quad \{not\ \neg a, not\ c\} \quad ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle \\
M_{12} & = & \langle \{not\ a, not\ \neg a, not\ c\} ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\} \rangle
\end{array}$$

and those resulting from the union of each of the ones above with the subsets of

$$\{not\ \neg b, not\ \neg c, not\ \neg d\}.$$

Note that there are A–models with assumptions $\{not\ b\}$ and with assumptions $\{not\ c\}$, but there is no A–model with assumptions $\{not\ b, not\ c\}$. This occurs because

$$P + \{not\ b, not\ c\}$$

is:

$$\begin{array}{rcl} \neg a & & \\ a & & \end{array} \qquad \begin{array}{rcl} b & \leftarrow & not\ d \\ c & \leftarrow & not\ d \end{array} \qquad \begin{array}{rcl} d & \leftarrow & not\ d \end{array}$$

which is clearly a contradictory program.

It is not the case that all programs have A–models. For example the program $P = \{a; \quad \neg a\}$ has no A–models. However it is clear that noncontradictory programs have at least one A–model (albeit $\langle\{\}; WFSX(P)\rangle$).

The notion of consistent A–models (definition 9.1.3) subsists for extended programs, i.e. $\langle A; M\rangle$ is consistent iff there exists no assumption $not\ L \in A$ such that $L \in M$. Note that there is no need to consider inconsistencies between objective literals. Those are taken care of by not considering A–models $\langle A; M\rangle$ where $P + A$ is contradictory.

Example 9.14 In the program of example 9.13 the A–models M_8, M_9, M_{10} and M_{12} are inconsistent. All others are consistent.

Even though the structural properties of A–models of normal programs do not subsist for extended ones, it is interesting to note that for noncontradictory programs the structural properties of consistent A–models are retained by extended programs. Moreover a program has consistent A–models iff it is noncontradictory.

Lemma 9.2.1 *Let $\langle A; \mathrm{WFSX}(P+A)\rangle$ be a consistent A–model of the extended logic program P. Then for every literal L :*

$$L \in \mathrm{WFSX}(P) \quad \Rightarrow \quad L \in \mathrm{WFSX}(P+A)$$

Proof: In appendix. $\diamondsuit$

Theorem 9.2.2 *An extended logic program P has at least one consistent A–model iff P is noncontradictory.*

Proof: ($\Leftarrow$) If the program is noncontradictory then clearly $\langle\{\}; \mathit{WFSX}(P)\rangle$ is a consistent A–model of P.

($\Rightarrow$) Let $\langle A; \mathit{WFSX}(P+A)\rangle$ be a consistent A–model of P. Then there is no pair $\{L, \neg L\}$ such that:

$$\{L, \neg L\} \subseteq \mathit{WFSX}(P+A)$$

So, by lemma 9.2.1, the same happens to $P + \{\}$, and thus P is noncontradictory. $\diamondsuit$

Theorem 9.2.3 *The set of all consistent A–models of an extended logic program P with the partial order $\leq_a$ is organized into a lower semilattice.*

Proof: In appendix. $\diamondsuit$

According to the CWA principles an assumption *not* L cannot be sustained if there is some consistent set of assumptions that defeats L. Note that, as for normal programs, also in extended programs the only way of defeating the assumption *not* L is by proving L. The objective complement of L, i.e. $\neg L$, only influences *not* L in order to prove it, not to defeat it. We do not even have to guarantee that mandatory default literals are indeed added since this is dealt with by WFSX[5]. So the definitions of defeating and sustainable A–models of normal programs (definitions 9.1.4 and 9.1.5, respectively) remain unchanged for extended programs.

It is easy to prove that lemma 9.1.2 subsists for extended programs. It is also clear from the definitions, that the A–model

$$\langle\{\}; \mathit{WFSX}(P)\rangle$$

is sustainable for every noncontradictory program P.

Given the structure of consistent A–models of extended programs and the proofs for normal programs, it follows almost directly that for extended programs the sustainable A–models are also organized into a lower semilattice.

Thus for extended programs, after taking care of inconsistent and unsustainable A–models, we are in a similar situation as with normal ones. Here, too, several maximal sustainable A–models might exist. The examples shown before (e.g. example 9.4) to illustrate this for normal programs can be used here for extended ones, since normal programs are a special case of extended ones. The program of example 9.13 is another case where several maximal sustainable A–models exist:

[5]Note that the meaning of an A–model $\langle A; \mathit{WFSX}(P+A)\rangle$ is $A \cup \mathit{WFSX}(P+A)$. Thus, given that WFSX imposes the mandatories (via coherence), these are guaranteed in the meaning of the A–model.

Example 9.15 In the program of example 9.13 all A–models with assumptions *not* a and *not* $\neg a$ are defeated, respectively by M_3 and M_2. Thus sustainable A–models are:

$$\begin{array}{lll} M_1 = & \langle \quad \{\} & ; \quad \{not\ \neg b, not\ \neg c, not\ \neg d\} \quad \rangle \\ M_2 = & \langle\ \{not\ b\} & ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\}\ \rangle \\ M_3 = & \langle\ \{not\ c\} & ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\}\ \rangle \end{array}$$

and the ones resulting from the union of each of the ones above with the subsets of

$$\{not\ \neg b, not\ \neg c, not\ \neg d\}.$$

Maximal sustainable A–models are, though:

$$\begin{array}{l} \langle\ \{not\ b, not\ \neg b, not\ \neg c, not\ \neg d\}\ ; \{\neg a, not\ a, not\ \neg b, not\ \neg c, not\ \neg d\}\ \rangle \\ \langle\ \{not\ c, not\ \neg b, not\ \neg c, not\ \neg d\}\ ; \{a, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\}\ \rangle \end{array}$$

In section 9.1.2, after having the lower semilattice of all sustainable A–models of a program P, we proceed to determine among those the ones that constitute the O–semantics and the O–model of P. This process was motivated by the principles that the CWA must be unique and maximal.

In order to achieve that, we devised a recursive peeling procedure on candidate structures that at each stage delete from the structure some non–preferred A-Models, the preference criterium being guided by the notion of tenability. Recall that tenability required that a maximal sustainable set of assumptions be not contradicted by the consequences of adding to it another competing maximal sustainable set.

For extended programs a set of assumptions can be contradicted by adding to it another competing one, if the resulting program either defeats the original set of assumptions (as for normal programs), or is contradictory. This last additional condition for untenability can be seen in the program of example 9.13:

Example 9.16 The maximal sustainable A–models shown in example 9.15 are both untenable because by adding the assumptions of them both to P the resulting program is contradictory. In fact:

$$P + \{not\ b, not\ c, not\ \neg b, not\ \neg c, not\ \neg d\} = \begin{array}{rcl} \neg a & & \\ a & & \\ b & \leftarrow & not\ d \\ c & \leftarrow & not\ d \\ d & \leftarrow & not\ d \end{array}$$

The rationale for saying that both the maximal sustainable A–models are untenable is grounded in that there is no reason to prefer one over the other, and they cannot be both assumed jointly. Thus in the end none of them is tenable.

Note the similarities between this and the method in chapter 8 for finding a unique revision of a contradictory program.

Accordingly, and retaining the definition 9.1.6 of candidate structure:

Definition 9.2.2 (Untenable A–models) *Let* $\{\langle A_k; M_k\rangle \mid k \in K\}$ *for* $k \in K$, *be the set of all maximal A–models in Candidate Structure* CS, *and let:*

$$P_J = P + \bigcup_{k \in K} A_j$$

An A–model $\langle A_i; M_i\rangle$ *is untenable wrt* CS *iff it is maximal in* CS *and either:*

- *P_J is contradictory, or*
- *there exists not $a \in A_i$ such that $a \in \mathrm{WFSX}(P_J)$.*

In order to define the retained candidate structure we first prove a lemma:

Lemma 9.2.4 *Let $\{\langle A_k; M_k\rangle \mid k \in K\}$ for $k \in K$, be the set of all maximal A–models in Candidate Structure CS.*

If none of those A–models is untenable then

$$\langle \bigcup_{k\in K} A_k; \mathrm{WFSX}(P + \bigcup_{k\in K} A_k)\rangle$$

is a sustainable A–model.

Proof: In appendix. ♢

The above lemma guarantees that when no untenable A–model exists in a candidate structure, its join can be safely added to it. Thus the definition of retained candidate structure for extended programs is similar to that for normal ones:

Definition 9.2.3 (Retained CS) *Let CS be a candidate structure, and let A be the union of all assumptions in all A–models of CS.*

The Retained Candidate Structure $R(CS)$ of CS is defined recursively in the following way:

- *$\langle A; \mathrm{WFSX}(P + A)\rangle \cup CS$ if there are no untenable A–models in CS.*
- *Otherwise, let Unt be the set of all untenable A–models wrt CS. Then*

$$R(CS) = R(CS - Unt)$$

Definition 9.2.4 (The O–semantics of extended programs) *The O–semantics of an extended logic program P is the retained candidate structure of the semilattice of all sustainable A–models of P.*

Let $\langle A; M\rangle$ be its maximal element. The O–model of P is $A \cup M$.

Example 9.17 The O–model of the program in example 9.13 is equal to its *WFSX*, i.e.

$$\{not\ \neg b, not\ \neg c, not\ \neg d\}$$

Now we prove some results that show that our goals for this section were indeed accomplished. One such goal was that the O–semantics of extended programs be a generalization of that for normal ones, i.e. if a program has no explicit negation then it is indifferent to use one definition or another:

Theorem 9.2.5 (Generalization of the O–semantics) *Let P be a normal logic program. The "O–semantics" of P coincides with the "O–semantics of extended programs" of P.*

Proof: Given that for normal programs WFS and *WFSX* coincide (cf. theorem 4.3.6), and that no normal program can be contradictory, it is easy to check that each of the definition modified in order to obtain the "O–semantics of extended programs" coincide with the original ones for normal programs. ♢

Another goal is that the O–semantics be defined for the same programs as *WFSX*:

Theorem 9.2.6 (Existence of the O–semantics) *An extended logic program P is noncontradictory iff it has O–semantics.*

Proof: By theorem 9.2.2 P is noncontradictory iff it has at least one consistent A–model. It is clear that P has at least one consistent A–model iff it has at least one sustainable A–model, albeit $\langle\{\}, WFSX(P)\rangle$ which, by definition, is always consistent and sustainable.

From lemma 9.2.4 and the definition of retained candidate structure it follows directly that P has sustainable A–models iff P has O–semantics. $\diamondsuit$

Finally, another goal was that O–semantics extend *WFSX* by adding to it some more assumptions:

Theorem 9.2.7 (Extension of *WFSX*) *Let P be a noncontradictory extended logic program, whose well–founded model according to* WFSX *is M, and whose O–model is O.*

Then $M \subseteq O$.

Proof: By lemma 9.2.1, for every consistent A–model $\langle A, WFSX(P + A)\rangle$:

$$WFSX(P) \subseteq WFSX(P + A) \qquad (*)$$

Since the O–model is by definition the meaning of one A–model, let $\langle A_O; WFSX(P + A_O)\rangle$ be that A–model.

Then, and using the result of $(*)$:

$$M = WFSX(P) \subseteq WFSX(P + A_O) \subseteq A_O \cup WFSX(P + A_O) = O$$

$\diamondsuit$

9.2.1 O–semantics or *WFSX* ?

At this point, and after having defined the O–semantics for extended programs, important questions have to be answered:

- *What semantics should be used as the base semantics for extended logic programs?*
- *Why was* WFSX *chosen as the preferred semantics in this work?*

First, it is not clear that it is always the case that more credulous semantics (i.e. assuming more hypotheses) are preferable to more sceptical ones. Intuitively, this is the case in the "game" example. But consider the following modification of the robot situation described in example 5.15:

Example 9.18 A robot is programmed to carry some money from bank 1 to bank 2. There are two possible routes, denoted a and b; the robot chooses one of them, provided that it has no reason to believe there is trouble along that route. If there is trouble on both routes then the robot signals "call repairs" (signal 1). If there is no reason to call repairs, the robot signals "no need of repairs" (signal 2). This task can be naturally formalized by the program (where the part of choosing one or the other route is omitted, since it can be found in example 5.15):

$$\begin{array}{lcl} signal(1) & \leftarrow & trouble(a), trouble(b) \\ signal(2) & \leftarrow & not\ signal(1) \end{array}$$

Now suppose that, additionally, the robot knows that if there is no reason to believe there is trouble in one of the routes, then there is trouble in the other. This can be expressed by the rules:

$$\begin{array}{lcl} trouble(a) & \leftarrow & not\ trouble(b) \\ trouble(b) & \leftarrow & not\ trouble(a) \end{array}$$

The well–founded model of the program constituted by these four rules is empty, while its O–model is:

$$\{not\ signal(1), signal(2)\}$$

Thus, according to the O–semantiocs the robot should signal "no need of repairs".

However, its knowledge about the troubles in the routes is compatible with having trouble in both of them[6], and so it can be arguable that the robot should be more sceptical, and be agnostic about signalling or not "no need of repairs".

The problem of this example is strongly related with the property of rationality[7]. Rationality is a cautious form of nonmonotonicity, and intuitively a semantics is rational if adding an atom A to a program P does not contradict the consequences of P alone, provided that $not\ A$ is not a consequence of it.

More formally, a semantics Sem is rational iff for every program P and every pair of atoms A and B :

$$\text{if } not\ A \notin Sem(P) \text{ and } B \in Sem(P) \text{ then } B \in Sem(P \cup \{A\})$$

In [Dix, 1992b] the author points out that O–semantics is not rational by showing a counterexample.

Example 9.19 Consider program P :

$$\begin{array}{lcl} c & \leftarrow & not\ b \\ b & \leftarrow & a \\ a & \leftarrow & not\ a \end{array}$$

The O–model of P contains c and does not contain $not\ a$, but the O–model of $P \cup \{a\}$ does not contain c.

Another property studied by Dix in [Dix, 1992b], and verified by WFS is "relevance". Intuitively a semantics is relevant iff the truth value of a literal is independent from the rules above it in the dependancy graph of the program (for a formal definition of relevance see section 10.1.2).

This property plays an important rôle if one wants to define procedures to decide if some literal belongs to the semantics. In particular, semantics that do not verify this property, cannot have top–down procedures for them, since in those semantics in order to decide the truth value of some literal L it might be necessary to examine rules that are not below L.

As proven by theorem 10.1.7 *WFSX* obeys relevance. This makes it amenable to the definition of top–down procedures. On the contrary, O–semantics for extended program is not relevant.

Example 9.20 Recall the program of example 9.13, whose O–model (calculated in example 9.17) is:

$$\{not\ \neg b, not\ \neg c, not\ \neg d\}$$

Since b is undefined in this program, one would expect that the same value would be assigned to it in the program just containing the rules with head b and the ones below those. This is in fact what would guarantee that a purely top–down method for deciding the truth value of b would be correct.

[6] Note that its knowledge just excludes the case of having both routes without problems.

[7] Rationality was introduced in [Kraus *et al.*, 1990] for nonmonotonic formalisms, and applied to normal and disjunctive logic programming, respectively in [Dix, 1991] and in [Dix, 1992a]. The generalization of this property for extended program can be found in section 10.1.1 below.

However, the program just containing the above mentioned rules is:

$$\begin{array}{lcl} b & \leftarrow & not\ d \\ d & \leftarrow & not\ d \end{array}$$

its O–model is $\{not\ b\}$, and thus the truth value of b is not the same.

This shows that computationally *WFSX* is preferable to O–semantics, i.e. it is easier to compute. Moreover the complexity of computing the *WFSX* of a datalog program is polynomial (as shown by theorems 10.1.9 and 10.1.10), and we conjecture that this is not the case for O–semantics, since it relies on first determining a class of models, and only after that choosing among those a preferred one.

Another issue that made us opt for *WFSX* is that it is sound wrt O–semantics (cf. theorem 9.2.7). Thus, even for the cases where O–semantics is the desired one, we can use the methods of *WFSX* to compute the consequences, and have the guarantee that the literals found are also consequences of the O–semantics. Note that the converse is not true, i.e. if the O–semantics is used for cases where *WFSX* is desired wrong answers are given.

Moreover O–semantics can be thought of as an addon to *WFSX*, inasmuch its very definition is in terms of such an addon. This way we can say that *WFSX* is the basic semantics, and its procedures are the basic ones. If in some cases O–semantics is desired, its computation can be performed relying on the procedures of *WFSX*. Note that this was also the spirit in contradiction removal, where instead of opting for some more sceptical semantics we have relied on *WFSX* and its procedures, and provide it with some addon mechanisms in order to produce the other more sceptical semantics.

Last, but not least, there is the historical nexus in the preparation of this work. From the start, the main goal was to extend logic programming with a second kind of negation, and to study especially the issues of the newly introduced negation, its benefits to nonmonnotonic reasoning formalisms, and its appropriateness for artificial intelligence problems. The choice of an already existing and widely accepted or understood semantics as a basis for the generalization, provided from the start for the easier acceptance of our work and ideas on explicit negations in the international scientific comunity.

The definition of a semantics assuming more negative hypotheses appeared after, and as an addon to the previously defined semantics. Its adoption from the start, because of its added complexity and new concepts, could jeopardize our primary goal.

Chapter 10

Further Properties and Comparisons

Throughout the previous chapters, several properties of *WFSX* were studied, and many comparisons with other semantics were made. Special importance was given to epistemic properties, and to comparisons based on epistemic arguments.

In this chapter we present some additional properties of *WFSX*, and make further comparisons with other semantics based on these properties, which are essentially structural in nature.

10.1 Properties of *WFSX*

Although most of the properties of *WFSX* presented up to now are of an epistemic nature, some structural properties too were already presented:

In section 4.3, it is shown that a least partial stable model – the well–founded model – always exists for noncontradictory programs (cf. theorem 4.3.1), and that that model can be obtained by an iterative bottom–up construction (cf. theorem 4.3.2 and definition 4.3.1). Moreover, we produced an iterative process for finding if a program is contradictory (cf. theorem 4.3.5). Also in that section, we prove that for normal programs the results of *WFSX* are equivalent to the results of the well–founded semantics of [Gelder *et al.*, 1991] (cf. theorem 4.3.6).

In section 5.1.3 some other properties of extended logic programs are brought out, namely: intrinsic consistency, coherence and supportedness. The proof that *WFSX* complies with the first two is trivial. The proof of the third is to be found below in section 10.1.2.

In section 6.3 some properties of Ω–default theories are exhibited and proven. Given the correspondence result of theorem 6.6.1, all these properties are verified by *WFSX* as well. In particular, *WFSX* complies with the property of modularity.

In section 6.7 an alternative definition of *WFSX* is given, and additional properties concerning it are supplied. Among these are several different iterative constructions for the well–founded model.

Via the equivalence result of theorem 7.4.9, all the properties presented in section 7.4 for complete scenaria semantics are also properties of *WFSX*. In particular, one such property points out that partial stable models under set inclusion are organized into a downward–complete semilattice (cf. point 1 of theorem 7.4.1), its least element being the well–founded model (cf. point 2 of the same theorem).

In order to make more formal comparisons between the various semantics for normal programs, in [Dix, 1991] the author submits some abstract properties a semantics should comply with. He begins by studying the application to normal logic program semantics of some struc-

tural properties defined for nonmonotonic reasoning formalisms in [Kraus *et al.*, 1990], and points out the importance, in normal programs, of properties such as *cumulativity* and *rationality*, that provide for a *cautious* form of nonmonotonicity.

More recently, in [Dix, 1992b], this author generalizes his previous work and presents an assortment of properties he claims must be obeyed by every *reasonable* semantics of normal programs. The motivation is to provide combinations of properties that guarantee a complete and unique characterization of a semantics via such properties. In this section we generalize some of the properties presented in [Dix, 1991, Dix, 1992b] for extended logic programs, and study whether *WFSX* complies with them.

Here too, we study the complexity of *WFSX*, and prove results needed for the proofs of previous theorems in this work.

The structure of this section is as follows: in section 10.1.1 we study structural properties related to the form of nonmonotonicity used by the semantics; then, in section 10.1.2, we study properties related to the form and transformations of programs; finally, in section 10.1.3 we prove some complexity results for *WFSX*.

10.1.1 Cumulativity and rationality

It is well known that semantics for logic programs with negation by default are nonmonotonic. However, some weak forms of monotonicity can still be verified by such semantics. Here we point out the importance of two such weak forms of monotonicity – cumulativity and rationality – for extended logic programs semantics, and examine whether *WFSX* complies with them.

Monotonicity imposes that for every program P and every pair of objective literals A and B of P

$$B \in Sem(P) \quad \Rightarrow \quad B \in Sem(P \cup \{A\})$$

In semantics of logic programs this property is not verified, and not even desired, for every such pair of objective literals. However, for some pairs, this property can be verified by some semantics, and in fact it can be very computationally useful.

One such case is when A is itself a consequence of P under the semantics Sem. The imposition of such a restriced form of monotonicity expresses that the addition of consequences of the semantics does not interfere with other consequences or, in other words, the consequences of a program, or lemmas, can safely be added to it. This weak form of monotonicity is usually called cumulativity.

Before defining cumulativity for extended logic programming we make a preliminary remark:

Remark 10.1.1 *The study of this kind of properties of logic programs is made in the sceptical version of a semantics (cf. [Dix, 1991]), i.e. $L \in Sem(P)$ is understood as: L belongs to all models determined by the semantics Sem when applied to the program P. Here this study is simplified since, by theorem 4.3.1, a literal belongs to all models of the semantics* WFSX *iff it belongs to the well–founded model. Thus, in the sequel we use $L \in \mathrm{WFSX}(P)$ to denote that L belongs to the well–founded model of P or, equivalently, to all partial stable models of P.*

The generalization of cumulativity for extended logic programs is quite straightforward: it is just a rephrasing of cumulativity for normal programs as it appears in [Dix, 1991], with the additional proviso that the program be noncontradictory:

Definition 10.1.1 (Cumulativity) *A semantics Sem is cumulative iff for every noncontradictory program P and any two objective literals A and B of P :*

$$\text{if } A \in Sem(P) \text{ and } B \in Sem(P) \text{ then } B \in Sem(P \cup \{A\})^1$$

This properties states that whenever an objective literal A has been derived from P, A can be used as a lemma and does not affect the set of objective literals derivable from P alone. If this condition is not valid, intermmediate lemmas are of no use. This indicates that noncumulative semantics may be computationally very expensive. As shown below, *WFSX* is a cumulative semantics, and so memoizing techniques can be used in its computation:

Theorem 10.1.1 *The* WFSX *semantics for extended logic programs is cumulative.*

Proof: We will prove that the complete scenaria semantics (definition 7.3.3) is cumulative. Given the equivalence between this semantics and *WFSX* (cf. theorem 7.4.9) this proves cumulativity for the latter.

Let $P \cup H$ be the least complete scenario of the noncontradictory program P. To prove this theorem, it is enough to show that if $P \cup H \vdash A$ and $P \cup H \vdash B$ then:

- $P \cup H \cup \{A\} \vdash B$;
- $P \cup H \cup \{A\}$ is the least complete scenario of $P \cup \{A\}$.

The proof of the first point is trivial since in the scenaria framework a scenario is a set of Horn clauses, and thus its consequences comply with monotonicity.

The proof of the second point is made in two steps. First we prove that $P \cup H \cup \{A\}$ is a complete scenario of $P \cup \{A\}$. Then we prove that there is no smaller complete scenario of $P \cup \{A\}$.

1. First we have to guarantee that $P \cup H \cup \{A\}$ is noncontradictory, i.e. it does not derive an objective literal L and its complement $\neg L$. Since $P \cup H \vdash A$, and $P \cup H$ is a set of Horn clauses, it follows clearly that the consequences of $P \cup H$ are the same of those of $P \cup H \cup \{A\}$. Given that $P \cup H$ is by hypothesis a complete scenario, it is also noncontradictory, and so the same happens with $P \cup H \cup \{A\}$.

   Furthermore, we have to show that every hypothesis in H is either mandatory or acceptable, and that all mandatory and acceptable hypotheses are in H.

   Recall that both the definitions of mandatory and acceptable are solely based on the consequences of the scenario.

   Again given that $P \cup H \vdash A$ and $P \cup H$ is a set of Horn clauses, the consequences of $P \cup H$ are the same of those $P \cup H \cup \{A\}$. Thus mandatory and acceptable hypotheses are the same for both $P \cup H$ and $P \cup H \cup \{A\}$, and given that the former is a complete scenario, the latter is also one.

2. The proof that it is the least scenario follows easily using the same arguments as in 1.

$\diamondsuit$

[1] This property is usually dubed "cautious monotonicity"(CM). In rigour, cumulativity stands for CM plus *Cut*, where this last property is defined by:

$$\text{if } A \in Sem(P) \text{ and } B \in Sem(P \cup \{A\}) \text{ then } B \in Sem(P)$$

Since all known semantics for both normal and extended programs trivially comply with Cut, it is equivalent to say that a semantics is cumulative, or that it complies with CM. Here, for the sake of generality, we use the term cumulativity.

In [Dix, 1991], the author presents another property – rationality – also related to cautious forms of nonmonotonicity. For normal logic programs this property is stronger than cumulativity, in the sense that every rational semantics is cumulative, but not vice–versa [2]. The straightforward generalization of this property for extended programs, following the same lines of that of cumulativity, is:

Definition 10.1.2 (Strong rationality) *A semantics Sem is strongly rational iff for every noncontradictory program P and any two objective literals A and B of P :*

$$\text{if } \mathit{not}\ A \notin \mathit{Sem}(P) \text{ and } B \in \mathit{Sem}(P) \text{ then } B \in \mathit{Sem}(P \cup \{A\})$$

The example below shows that *WFSX* is not strongly rational:

Example 10.1 Consider program P :

$$\begin{array}{rcl} \neg b & & \\ b & \leftarrow & a \\ a & \leftarrow & \mathit{not}\ a \end{array}$$

For this program $\mathit{not}\ a \notin \mathit{WFSX}(P)$ and $\neg b \in \mathit{WFSX}(P)$. However the program $P \cup \{a\}$ is contradictory, and so $\neg b \notin \mathit{WFSX}(P \cup \{a\})$.

At this point we would like to recall the rationale behind rationality. While cumulativity expresses that the addition of some consequences of the semantics do not interfere with the other consequences, rationality expresses that the addition of literals that are *compatible* with the program does not interfere with its consequences.

For normal logic programs an atom A is compatible with a program P iff its negation $\mathit{not}\ A$ is not in the semantics of P. Note that the same does not happen for extended programs. For instance, in the program of example 10.1 $\mathit{not}\ a$ is not a consequence of the semantics, but a is not compatible with the program.

In extended programs, and in order to guarantee that some objective literal L is compatible with a program P, we have not only to verify that $\mathit{not}\ L$ is not a consequence of P, but also that the program obtained by adding L to P is noncontradictory. This suggests a weaker version of the rationality for extended programs that, in our view, is closer to the original rationale of rationality for normal programs:

Definition 10.1.3 (Cautious rationality) *A semantics Sem is cautiously rational iff for every noncontradictory program P and any two objective literals A and B of P, if $\mathit{not}\ A \notin \mathit{Sem}(P)$, and $P \cup \{A\}$ is a noncontradictory program, and $B \in \mathit{Sem}(P)$, then:*

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

Theorem 10.1.2 *The* WFSX *semantics for extended logic programs is cautiously rational.*

Proof: As in the proof of theorem 10.1.1, here we also prove the property for *WFSX* via its equivalence to complete scenaria semantics.

For simplicity, and without loss of generality (cf. corollary 10.1.2), we assume that programs are in the semantic kernel form, i.e. no objective literal appears in the body of rules.

Let P be a noncontradictory program in that form, let $P \cup H$ be its least complete scenario, and let A and B be two objective literals of P such that:

$$\begin{array}{ll} (i) & \mathit{not}\ A \notin H \\ (ii) & P \cup \{A\} \text{ is noncontradictory} \\ (iii) & P \cup H \vdash B \end{array}$$

We begin by proving that:

[2] For example the O–semantics of normal logic programs is not rational (cf. example 9.19) but is cumulative (cf. [Dix, 1992c]).

1. if *not* L is mandatory in $P \cup H$, it is also mandatory in $P \cup H \cup \{A\}$.

   Given that $P \cup H$ is a complete scenario, it contains all its mandatories. Thus *not* L is mandatory iff $P \cup H \vdash \neg L$. Given that scenaria are sets of Horn clauses, $P \cup H \cup \{A\} \vdash \neg L$, and so, by definition of mandatory, *not* L is mandatory in $P \cup H \cup \{A\}$.

2. if *not* L is acceptable in $P \cup H$ it is also acceptable in $P \cup H \cup \{A\}$.

   By definition of acceptable hypothesis, *not* L is acceptable in $P \cup H$ iff

   $$\forall E, P \cup E \vdash L \Rightarrow \exists not\ L' \in E \mid P \cup H \vdash L'$$

   Again given that a scenario is a set of Horn clauses, its consequences are monotonic, and so the above formula entails that:

   $$\forall E, P \cup E \vdash L \Rightarrow \exists not\ L' \in E \mid P \cup H \cup \{A\} \vdash L'$$

   By condition (i) it follows that *not* A is not acceptable in $P \cup H$. Thus we can assume in the formula above that L is different from A. Given that by hypotheses the program is in the semantic kernel form, for every objective literal L different from A :

   $$P \cup E \vdash L \quad \Leftrightarrow \quad P \cup E \cup \{A\} \vdash L$$

   So, if *not* L is acceptable in $P \cup H$ then:

   $$\forall E, P \cup E \cup \{A\} \vdash L \Rightarrow \exists not\ L' \in E \mid P \cup H \cup \{A\} \vdash L'$$

   i.e., by definition of acceptable, *not* L is acceptable in $P \cup H \cup \{A\}$.

By condition (iii), and given that consequences of a scenario are monotonic, it follows that

$$P \cup H \cup \{A\} \vdash B$$

Since, by points 1 and 2 above, mandatory and acceptable hypotheses subsist in $P \cup H \cup \{A\}$, and consistency is guaranteed by condition (ii), it follows that the least complete scenario of $P \cup \{A\}$ is of the form:

$$P \cup H' \cup \{A\}$$

where $H' \supseteq H$.

Thus $P \cup H' \cup \{A\} \vdash B$, i.e. $B \in WFSX(P \cup \{A\})$. $\diamondsuit$

10.1.2 Partial evaluation and relevance

Here we study properties related to the form of programs, and with the preservation of the semantics when some transformations are applied to programs.

One such important property is the so called *principle of partial evaluation* [Dix, 1992b]. This principles states that the semantics of every program should be preserved under unfolding of objective literals. The example below shows that *WFSX* is not preserved under the usual unfolding techniques[3] for normal programs:

[3] In this work we do not give a formal definition of unfolding for normal programs, and assume that this is known to the reader.

Example 10.2 Recall program P of example 4.9:

$$\begin{array}{lll} c \leftarrow a & \quad & a \leftarrow b \\ \neg a \leftarrow & & b \leftarrow not\ b \end{array}$$

whose *WFSX* is:

$$\{\neg a, not\ a, not\ c, not\ \neg b, not\ \neg c\}$$

By unfolding the objective literal a in the rule for c we obtain the program P' :

$$\begin{array}{lll} c \leftarrow b & \quad & a \leftarrow b \\ \neg a \leftarrow & & b \leftarrow not\ b \end{array}$$

whose *WFSX* is:

$$\{\neg a, not\ a, not\ \neg b, not\ \neg c\}$$

Note that the truth value of c is not preserved.

This happens because the unfolding of a did not take into account the fact that $\neg a$ is a consequence of the program. In order to define an unfolding technique for extended logic programs care must be taken in such cases. One has to guarantee that the unfolding of some atom A does not interfere with the fact that $\neg A$ belongs to the consequences of the program.

We shall see that one way of guaranteeing this is by adjoining to objective literal L the default literal $not\ \neg L$, before using the usual techique for unfolding L. Note that program P'' :

$$\begin{array}{lll} c \leftarrow not\ \neg a, b & \quad & a \leftarrow b \\ \neg a \leftarrow & & b \leftarrow not\ b \end{array}$$

has indeed the same *WFSX* of program P.

In order to define the unfolding technique for extended programs we first prove the theorem:

Theorem 10.1.3 *Let P be any extended logic program, and let P' be the a program obtained from P by adding to the body of some rule:*

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

the default literal $not\ \neg B_i$, where $1 \leq i \leq n$ and $\neg B_i$ denotes the objective complement of B_i.

Then:

- *M is a PSM of P iff M is a PSM of P'.*
- *P is contradictory iff P' is contradictory.*

Proof: In appendix. $\diamondsuit$

From this theorem there follows an important corollary, already used above in this work (e.g. in the definition of scenaria):

Corollary 10.1.1 *For every program P and its canonical program P'*

$$\mathrm{WFSX}(P) = \mathrm{WFSX}(P')$$

Proof: Follows directly from the theorem and the definition 2.1.1 of canonical program. $\diamondsuit$

Let us define now the principle of partial evaluation for extended programs:

Definition 10.1.4 (Principle of partial evaluation) *Let P be an extended logic program, and let:*

$$\begin{array}{rcl} L & \leftarrow & BodyL_1 \\ & \vdots & \\ L & \leftarrow & BodyL_n \end{array}$$

be all rules of P with head L. Assume further that $BodyL_1, \ldots, BodyL_n$ do not contain L.

We denote by $unfold(P, L)$ the program obtained from P by replacing each rule $H \leftarrow L, BodyH$ (i.e. each rule with L in the body) by:

$$\begin{array}{rcl} H & \leftarrow & not\ \neg L, BodyL_1, BodyH \\ & \vdots & \\ H & \leftarrow & not\ \neg L, BodyL_n, BodyH \end{array}$$

The principle of partial evaluation states that the semantics of P is equal to the semantics of $unfold(P, L)$.

Theorem 10.1.4 WFSX *complies with the principle of partial evaluation.*

Proof: Let $P' = unfold(P, L)$.

Recall that, according to theorem 6.7.1, $T \cup not\ F$ is a PSM of a program P iff

$$\begin{array}{rcl} T & = & \Gamma_P \Gamma_{P_s} T \\ T & \subseteq & \Gamma_{P_s} T \\ F & = & \{L \mid L \notin \Gamma_{P_s} T\} \end{array}$$

and that $\Gamma_P S$ is the least Herbrand model of the positive program ${\frac{P}{S}}^{gl}$ obtained by deleting from P all rules with a literal *not* A in the body such that $A \in S$, and then deleting all default literals from the body of the remaining rules.

We begin by proving that for any set of objective literals S :

$$\Gamma_{P_s} S = \Gamma_{P'_s} S \qquad (*)$$

If $\neg L \notin S$ then the default literals *not* $\neg L$ introduced by the partial evaluation are deleted in ${\frac{P'}{S}}^{gl}$, and so this program is obtainable from ${\frac{P}{S}}^{gl}$ via unfolding of L. Given that unfolding preserves the least Herbrand model of a positive program, $\Gamma_{P_s} S = \Gamma_{P'_s} S$.

If $\neg L \in S$ then the only possible difference between the ${\frac{P}{S}}^{gl}$ and ${\frac{P'}{S}}^{gl}$ is that rules with *not* $\neg L$ in the body are deleted in the latter but not in the former. Given that the program is seminormal, by definition all rules with head L are deleted in both positive programs.

The rules that remain in ${\frac{P}{S}}^{gl}$ and are deleted in ${\frac{P'}{S}}^{gl}$, have in the former the objective literal L in their bodies. Thus, since no rules for L exist in ${\frac{P}{S}}^{gl}$, the remaining rules are useless to determine the least Herbrand model of that program, and so $\Gamma_{P_s} S = \Gamma_{P'_s} S$.

Now, let us assume that $T \cup not\ F$ is a PSM of P. Then $T = \Gamma_P \Gamma_{P_s} T$. By (*):

$$T = \Gamma_P \Gamma_{P'_s} T$$

If $\neg L \notin \Gamma_{P'_s} T$ then the default literals *not* $\neg L$ introduced by the partial evaluation are deleted in ${\frac{P'}{\Gamma_{P'_s} T}}^{gl}$, and so this program is obtainable from ${\frac{P}{\Gamma_{P_s} T}}^{gl}$ via unfolding of L. Thus, for the same reasons as before, $\Gamma_P \Gamma_{P'_s} T = \Gamma_{P'} \Gamma_{P'_s} T$.

If $\neg L \in \Gamma_{P'_s} T$ then $L \notin T$, since otherwise L would be true in the PSM and $\neg L$ undefined, which is impossible because every PSM complies with coherence. So the rules that are deleted in $\frac{P'}{\Gamma_{P'_s} T}^{gl}$ but not in $\frac{P}{\Gamma_{P_s} T}^{gl}$ are useless to determine the least Herbrand model (for the same reasons as before) and thus $\Gamma_P \Gamma_{P'_s} T = \Gamma_{P'} \Gamma_{P'_s} T$.

So:

$$T = \Gamma_{P'} \Gamma_{P'_s} T$$

Directly from $(*)$ it follows that:

$$\begin{array}{rcl} T & \subseteq & \Gamma_{P'_s} T \\ F & = & \{L \mid L \notin \Gamma_{P'_s} T\} \end{array}$$

Thus $T \cup not\ F$ is a PSM of P'.

The proof that if $T \cup not\ F$ is a PSM of P' then it is also a PSM of P, is quite similar to the one above and is omitted for brevity. $\diamondsuit$

Another property presented in [Dix, 1992b] is equivalence. It is especially important in this work because, together with the partial evaluation principle, it allows us to prove a result that has been widely used to simplify the proofs of theorems throughout this work.

Definition 10.1.5 (Equivalence) *Let P' be the extended logic program obtained from P by deleting every rule:*

$$L \leftarrow L, Body$$

i.e. every rule whose head is contained in the body.

Equivalence states that the semantics of P' is equal to the semantics of P.

Theorem 10.1.5 WFSX *complies with equivalence.*

Proof: Given the equivalence between this semantics and *WFSX* (cf. theorem 7.4.9), we prove that the complete scenaria semantics (definition 7.3.3) complies with equivalence.

By definition, scenaria are sets of Horn clauses, and rules of the form $L \leftarrow L, Body$ result in tautologies in the scenaria framework. Thus for any program P, any set of hypotheses H and any objective literal A :

$$P \cup \{L \leftarrow L, Body\} \cup H \vdash A \quad \Leftrightarrow P \cup H \vdash A \qquad (*)$$

So, by their respective definitions, it follows directly that for every hypothesis $not\ A$:

- $not\ A \in Mand(P \cup H)$ iff $not\ A \in Mand(P \cup \{L \leftarrow L, Body\} \cup H)$.
- $not\ A \in Acc(P \cup H)$ iff $not\ A \in Acc(P \cup \{L \leftarrow L, Body\} \cup H)$.

By definition of complete scenario:

$$P \cup H \text{ is a complete scenario of } P \ \Leftrightarrow \ H = Mand(P \cup H) \cup Acc(P \cup H)$$

By the results above, $H = Mand(P \cup H) \cup Acc(P \cup H)$ iff

$$H = Mand(P \cup \{L \leftarrow L, Body\} \cup H) \cup Acc(P \cup \{L \leftarrow L, Body\} \cup H)$$

Again, by definition of complete scenario:

$$\begin{array}{c} H = Mand(P \cup \{L \leftarrow L, Body\} \cup H) \cup Acc(P \cup \{L \leftarrow L, Body\} \cup H) \\ \Leftrightarrow \\ P \cup \{L \leftarrow L, Body\} \cup H \text{ is a complete scenario of } P \cup \{L \leftarrow L, Body\} \end{array}$$

Thus the complete scenaria of P are the same as the complete scenaria of

$$P \cup \{L \leftarrow L, Body\}.$$

By $(*)$ it follows also that the consequences of those scenaria are the same in both programs. $\diamondsuit$

Given the results of theorems 10.1.4 and 10.1.5, we next define a bottom–up process that transforms every extended program into another with no objective literals in the body of rules, and with the same *WFSX*.

Intuitively, in order to obtain such a transformed program, we begin by recording all rules with no objective literals in the body (hereafter called rules in the desired form). Then we unfold all literals such that all of its rules are in the desired form. By performing this partial evaluation more rules become of that form. The process is iterated until a fixpoint is reached.

In order to formally define this process we begin with some preliminary definitions:

Definition 10.1.6 *Let P be an extended logic program. We define:*

- *$sk\_lits(P)$ is the set of objective literals L such that there is no rule in P with head L and with objective literals in the body.*
- *$sk\_rules(P)$ is the set of all rules in P such that their heads belong to $sk\_lits(P)$.*

Definition 10.1.7 (Semantic kernel transformation) *Let P and P' be two extended logic programs with the same Herbrand base, such that P' does not contain any objective literal in the body of its rules.*

Additionally, let $heads(P')$ be the set of all objective literals in the head of some rule of P', and let P_r be the program obtained from P by first deleting from it every rule whose head is in $heads(P')$, and then making the union of the result with P'.

We define:

$$SK_P(P') = P' \cup sk\_rules(unfold(P_r, heads(P')))$$

The semantic kernel transformation SK_P of program P is the least fixpoint of the sequence:

$$\begin{aligned} P_0 &= sk\_rules(P) \\ P_{\alpha+1} &= SK_P(P_\alpha) \end{aligned}$$

Theorem 10.1.6 *The semantic kernel transformation SK_P of an extended program P uniquely exists, and is in the semantic kernel form, i.e. it is a set of rules with no objective literal in their bodies.*

Moreover the WFSX *of SK_P is equal to the* WFSX *of P.*

Proof: The existence, uniqueness, and semantic kernel form of SK_P are guaranteed by its construction.

The *WFSX* equivalence with the program P follows easily from the fact that the transformation is solely based on partial evaluations, and that the rules that are never added are clearly those that for some partial evaluation their head is contained in the body. Thus theorems 10.1.4 and 10.1.5 guarantee such an equivalence. $\diamondsuit$

From this theorem it follows directly that:

Corollary 10.1.2 *For every program P there exists one program P' with no objective literals in the body of its rules, such that the* WFSX *of P is equal to the* WFSX *of P'.*

Example 10.3 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & \neg b, not\ c \\ \neg b & \leftarrow & d, not\ e \\ \neg b & \leftarrow & not\ p \\ d & \leftarrow & f \\ f & & \end{array} \qquad \begin{array}{rcl} p & \leftarrow & q \\ q & \leftarrow & p, not\ c \end{array}$$

and let us calculate SK_P.

- $sk\_rules(P) = \{f\}$, and so $P_0 = \{f\}$. Note that $\neg b \leftarrow not\ p$ does not belong to P_0. This is because there is another rule with head $\neg b$ and with an objective literal in its body.
- By unfolding f in P (cf. definition 10.1.4) we obtain:

$$\begin{array}{rcl} a & \leftarrow & \neg b, not\ c \\ \neg b & \leftarrow & d, not\ e \\ \neg b & \leftarrow & not\ p \\ d & \leftarrow & not\ \neg f \\ f & & \end{array} \qquad \begin{array}{rcl} p & \leftarrow & q \\ q & \leftarrow & p, not\ c \end{array}$$

  and thus $P_1 = \{d \leftarrow not\ \neg f;\quad f\}$.
- By unfolding both d and f in the program, using their rules in P_1 we obtain:

$$\begin{array}{rcl} a & \leftarrow & \neg b, not\ c \\ \neg b & \leftarrow & not\ \neg d, not\ \neg f, not\ e \\ \neg b & \leftarrow & not\ p \\ d & \leftarrow & not\ \neg f \\ f & & \end{array} \qquad \begin{array}{rcl} p & \leftarrow & q \\ q & \leftarrow & p, not\ c \end{array}$$

  So $P_2 = P_1 \cup \{\neg b \leftarrow not\ \neg d, not\ \neg f;\quad \neg b \leftarrow not\ p\}$.
- By also unfolding $\neg b$ we get:

$$\begin{array}{rcl} a & \leftarrow & not\ b, not\ \neg d, not\ \neg f, not\ e, not\ c \\ a & \leftarrow & not\ b, not\ p, not\ c \\ \neg b & \leftarrow & not\ \neg d, not\ \neg f, not\ e \\ \neg b & \leftarrow & not\ p \\ d & \leftarrow & not\ \neg f \\ f & & \end{array} \qquad \begin{array}{rcl} p & \leftarrow & q \\ q & \leftarrow & p, not\ c \end{array}$$

  and thus:

$$P_3 = P_2 \cup \left\{ \begin{array}{rcl} a & \leftarrow & not\ b, not\ \neg d, not\ \neg f, not\ e, not\ c \\ a & \leftarrow & not\ b, not\ p, not\ c \end{array} \right\}$$

- It is easy to see that $P_4 = P_3$.

Thus SK_P is the program:

$$\begin{array}{rcl} a & \leftarrow & not\ b, not\ \neg d, not\ \neg f, not\ e, not\ c \\ a & \leftarrow & not\ b, not\ p, not\ c \\ \neg b & \leftarrow & not\ \neg d, not\ \neg f, not\ e \\ \neg b & \leftarrow & not\ p \\ d & \leftarrow & not\ \neg f \\ f & & \end{array}$$

Note that in fact $WFSX(P) = WFSX(SK_P)$.

Relevance is another property of semantics, related with transformations over programs, and also studied in [Dix, 1992b] for comparing semantics of normal logic program. Intuitively, a semantics complies with relevance iff the truth value of any literal in it is determined by the rules on which that literal depends. In order to formalize this notion we first define the dependency relation:

Definition 10.1.8 (Dependency relation) *An objective literal A depends on a literal L in an extended logic program P iff $L = A$ or there is a rule in P with head A and L' in its body and L' depends on L.*

A default literal not A depends on a literal L in P iff $L = not\ A$, $L = \neg A$ or there is a rule in P with head A and not L' in the body and not L' depends on L. Here, by $\neg A$ (resp. not L') we mean the objective (resp. default) complement of A (resp. L').

By $dep\_on(A, P)$ we mean the set of all literals L such that A depends on L.

Example 10.4 Consider program P :

$$\begin{array}{llllllll} (1) & a & \leftarrow & b, not\ c & c & \leftarrow & d, not\ e & (3) \\ (2) & \neg c & \leftarrow & not\ g & e & \leftarrow & f & (4) \end{array}$$

The reader can check that, for example:

$$\begin{array}{rcl} dep\_on(a, P) & = & \{a, b, not\ c, \neg c, not\ g, \neg g, not\ d, \neg d, e, f\} \\ dep\_on(not\ a, P) & = & \{not\ a, \neg a, not\ b, \neg b, c, d, not\ e, \neg e, not\ f, \neg f\} \\ dep\_on(b, P) & = & \{b\} \\ dep\_on(not\ b, P) & = & \{not\ b, \neg b\} \\ dep\_on(c, P) & = & \{c, d, not\ e, not\ f\} \\ dep\_on(not\ c, P) & = & \{not\ c, \neg c, not\ g, not\ d, \neg d, e, f\} \end{array}$$

Definition 10.1.9 (Relevant rules) *The set of relevant rules of program P for literal L, $rel\_rul(P, L)$, is the set of all rules with head H such that $H \in dep\_on(L, P)$ or not $H \in dep\_on(L, P)$.*

Example 10.5 For program P of example 10.4, the set of relevant rules for the literals whose dependencies were calculated there, are (where for brevity only their identifying numbers are presented):

$$\begin{array}{rcl} rel\_rul(P, a) & = & \{(1), (2), (3), (4)\} \\ rel\_rul(P, not\ a) & = & \{(1), (3), (4)\} \\ rel\_rul(P, b) & = & \{\} \\ rel\_rul(P, not\ b) & = & \{\} \\ rel\_rul(P, c) & = & \{(3), (4)\} \\ rel\_rul(P, not\ c) & = & \{(2), (3), (4)\} \end{array}$$

Definition 10.1.10 (Relevance) *A semantics Sem complies with the principle of relevance iff for every noncontradictory program P and every literal L*

$$L \in Sem(P) \quad \Leftrightarrow \quad L \in Sem(rel\_rul(P, L))$$

The importance of this structural property is well recognizable if we think of top–down procedures for deciding the truth value of some literal. A semantics not complying with this principle cannot have a purely top–down procedure based on rewriting techiques.

Theorem 10.1.7 WFSX *complies with the principle of relevance.*

Proof: It is easy to see that for the definition of support (definition 8.3.14) of some literal L in any program P, only rules of $rel\_rul(P, L)$ are used. Since the truth value of a literal can be determined from the existence or not or a support for it (cf. proposition 8.3.10), it follows easily that *WFSX* complies with relevance. $\diamondsuit$

Another property mentioned above in this work (in section 5.1.3) is supportedness. Recall that a semantics complies with supportedness if an objective literal L is true in the semantics of P iff there is rule in P with head L and whose body is also true in the semantics of P.

Theorem 10.1.8 WFSX *complies with supportedness.*

Proof: Trivial in the complete scenario semantics (which is equivalent to *WFSX* by theorem 7.4.9). $\diamondsuit$

10.1.3 Complexity results

Several times in this work we've said that we are interested in a computable semantics, and that computational cost is for us an important issue.

Unfortunately *WFSX* is not recursively enumerable (cf. definition 4.3.1). This is a difficulty *WFSX* shares with most reasonable semantics for normal logic programs, including the well–founded semantics (WFS) of [Gelder *et al.*, 1991].

However, as proven in [Gelder *et al.*, 1991], the complexity of the decision problem in WFS for Datalog programs (i.e. programs without function symbols) is polynomial. In this section we show that the addition of explicit negation into WFS does not increase the complexity of the latter.

We begin by showing that if one knows à priori that some Datalog program P is noncontradictory then the decision problem[4] in *WFSX* of P is polynomial.

Theorem 10.1.9 *The decision problem for any noncontradictory Datalog program P under* WFSX *is polynomial in the size of the ground version of P.*

Proof: This proof follows closely the proof about the complexity of WFS in [Gelder *et al.*, 1991].

We show that the well–founded model can be constructed in polynomial time, after which any query can be answered immediately. We do this proof using the equivalent definition of *WFSX*, of theorem 6.7.2.

According to that theorem, the positive part of the well–founded model T is the least fixpoint of the operator $\Gamma\Gamma_s$, the negative part F being the complement of the application of Γ_s to that least fixpoint.

At each stage T_α of the induction, until the fixpoint is reached, at least one element of the Herbrand base is added to $T_{\alpha+1}$, so the fixpoint must be reached in a number of steps polynomial in the size of the $\mathcal{H}$[5]. So we need only show that $\Gamma\Gamma_s T_\alpha$ can be found in polynomial time and that, given T, F can also be found in polynomial time.

It is clear that for these proofs it is enough to show that, for any set S of objective literals, the computation of both ΓS and $\Gamma_s S$ is polynomial. Since $\Gamma_s S$ is equal to ΓS applied to a seminormal version of the program, and clearly the seminormal version is computable in linear time, we only show that the computation of ΓS is polynomial.

- The computation of ΓS starts by deleting all rules whose body contains a default *not* L such that $L \in S$. It is clear that this computation is $O(\mid S \mid * \mid P \mid)$.

[4] As usual, by decision problem we mean the problem of deciding whether some literal L belongs to the semantics of the program.

[5] This kind of argument is standard, viz. [Chandra and Harel, 1982, Vardi, 1982, Gurevich and Shelah, 1986, Immerman, 1986, Gelder *et al.*, 1991].

- Then all default literals in the bodies of the remaining rules are deleted. This computation is $O(\mid P \mid)$.
- Finally, the T_P of the resulting positive program is computed. It is well known that the computation of T_P of a positive program is polynomial.

Thus the computation of ΓS is polynomial. $\diamondsuit$

According to this theorem we can only say that if one knows that some program P is noncontradictory then it can be decided in polynomial time whether some literal is true in $WFSX(P)$. However the result can be generalized by withdrawing the à priori knowledge about the noncontradiction of P. This is so because:

Theorem 10.1.10 *The problem of determining whether a Datalog extended program P is contradictory under* WFSX *is polynomial in the size of the ground version of P.*

Proof: From the correspondence theorem 6.6.1 and proposition 6.3.2 it follows directly that a Datalog program P is contradictory iff the least fixpoint of the sequence:

$$\begin{aligned} T_0 &= \{\} \\ T_{\alpha+1} &= \Gamma\Gamma_s(T_\alpha) \end{aligned}$$

contains some objective literal L and its complement $\neg L$.

Since the computation of that fixpoint is polynomial (cf. theorem 10.1.9), it follows easily that to determine whether P is contradictory is also polynomial. $\diamondsuit$

10.2 Comparisons

Throughout the text above, several comparisons were made between *WFSX* and other semantics for extended logic programs.

Comparisons with the semantics of [Przymusinski, 1990a] were made in chapter 3 and section 5.1 where we argued that this semantics does not impose any connection between the two types of negations. In fact, as mentioned in chapter 3, our insatisfaction with the semantics of [Przymusinski, 1990a] in what concerns that desired connection was one of the main motivations for defining a new semantics for extended logic programs.

Also in section 5.1, some comparisons were made with the semantics of [Przymusinski, 1991b]. There we point out that that semantics does not comply with supportedness. Epistemic comparisons with that semantics were made not only in that very section, where we argued that supportedness closely relates to the use of logic as a programming language, but also in section 5.2 where we related the use of classical negation $\sim L$ of [Przymusinski, 1991b] with the epistemic reading *"L is not known to be true"*. In contradistinction, explicit negation $\neg L$ of *WFSX* has the reading *"L is known to be false"*. In subsection 5.2.2 we compared these two readings and argued in favour of the latter.

Epistemic comparisons with answer–set semantics [Gelfond and Lifschitz, 1990] were drawn indirectly in section 5.2 (via the correspondence between answer–set semantics and Moore's autoepistemic logic), and in chapter 6 (via the correspondence between answer–set semantics and Reiter's default logic).

However no detailed comparisons between *WFSX* and answer–set semantics concerning structural properties were made yet. The only structural properties pointed out for answer–sets were the ones studied in section 5.1, where we found out that intrinsic consistency, coherence and supportedness are verified by both answer–sets and *WFSX*.

Recall that, as mentioned in chapter 3, one of our main qualms with answer–set semantics was in what regards its structural and computational properties. In this section we make additional comparisons between *WFSX* and answer–sets. These comparisons are made either using the properties in the previous section, or via structural properties of nonmonotonic formalisms that correspond to answer–sets.

We start by comparing the complexity results of both semantics. In the previous section we have shown that for Datalog programs the complexity of both the decision problem and the problem of finding if some program is contradictory in *WFSX* is polynomial. In contrast, in [Marek and Truszczynski, 1991] the authors show that, even for Datalog programs, the problem of finding if a program has answer–sets is NP–complete, and the decision problem for programs with answer–sets is co–NP–complete.

As proven above, *WFSX* enjoys some structural properties with regard to the organization of its models. In particular:

- partial stable models under set inclusion are organized into a downward–complete semi-lattice, its least element being the well–founded model;
- the intersection of all partial stable models is equal to the well–founded model, and can be computed by an iterative bottom–up process.

None of these properties is enjoyed by answer–set semantics. In fact, by its very definition, no answer–set is comparable (wrt $\subseteq$) with other answer–sets. Thus, for deciding if some literal is a consequence of a program under the answer–set semantics one cannot rely on a single least model (as in *WFSX*) and, in contrast, have first to compute all answer–sets and then their intersection.

Given that answer–set semantics corresponds to Reiter's default logic (cf. [Gelfond and Lifschitz, 1990]), this problem is related with the property of uniqueness of minimal extension studied in section 6.2. There we point out more problems with Reiter's default logic (and given the correspondence results of [Gelfond and Lifschitz, 1990], also with answer–set semantics) that result from the inexistence of a unique minimal extension. In particular, we argue it is undesirable that the *cautious (or sceptical) version* of the semantics not be itself a model of it. Next we present some other undesirable properties of the sceptical version of answer–set semantics.

By the sceptical version of the answer–set semantics we mean (as usual) the semantics $AS(P)$ determined by:

$$\begin{array}{rll} L \in AS(P) & \text{iff} & L \text{ is in all answer-sets of } P \\ not\ L \in AS(P) & \text{iff} & \text{there is no answer–set of } P \text{ containing } L \end{array}$$

where L is any objective literal of the extended program P.

Cumulativity is one structural property obeyed by *WFSX* (cf. theorem 10.1.1) and not by the sceptical version of answer–sets. The example below shows this is indeed the case:

Example 10.6 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ c & \leftarrow & not\ a \\ c & \leftarrow & not\ c \end{array}$$

whose only answer–set is $\{c, b\}$. Thus $c \in AS(P)$, and $b \in AS(P)$. However

$$b \notin AS(P \cup \{c\}).$$

In fact $P \cup \{c\}$ has two answer–sets:

$$\{p, a\} \quad \text{and} \quad \{p, b\}$$

Since one of them does not contain b, $b \notin AS(P \cup \{c\})$.

This very same example also shows that answer–set semantics is neither strongly nor cautiously rational. In fact *not* $c \notin AS(P)$, $b \in AS(P)$, and $P \cup \{c\}$ is noncontradictory, but $b \notin AS(P \cup \{c\})$.

Being noncumulative, answer–set semantics not only gives in some cases very unintuitive results, but also some added problems in its computation accrue. In particular, even for propositional programs, the computation of answer-sets cannot be made by approximations[6]: once it is found that an objective literal is in every answer–set, that literal cannot be added as a fact to the program.

This also points out problems in finding an iterative bottom–up process for computing answer–set semantics, since usually such methods use already computed results as lemmas.

Another structural property studied in the previous section and obeyed by *WFSX* is relevance. The example below shows that answer–set semantics does not comply with relevence.

Example 10.7 Consider program P :

$$\begin{array}{rcl} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \\ c & \leftarrow & not\ a \\ \neg c & & \end{array}$$

whose only answer–set is $\{\neg c, a\}$. The rules relevant for a are the first two. However a is not in the answer–set semantics of just those relevant rules.

In fact, $rel\_rul(P, a)$ has two answer–sets: $\{a\}$, and $\{b\}$. Since one of them does not contain a, $a \notin AS(rel\_rul(P, a))$.

This shows that, in contradistinction with *WFSX*, there can be no purely top–down procedure for determining if some literal is true under the answer–set semantics. Such a procedure would have to examine more rules than the ones on which the literal depends.

Another interesting result concerning comparisons between *WFSX* and answer–sets is:

Theorem 10.2.1 *If an extended logic program has at least one answer–set it has at least one partial stable model.*

Proof: Follows directly from theorem 6.4.4, given the correspondence between Ω–extensions and PSMs (cf. theorem 6.6.1), and the correspondence between Reiter's extensions and answer–sets (cf. [Gelfond and Lifschitz, 1990]). $\diamondsuit$

From this theorem it follows that *WFSX* gives semantics to at least the same programs answer–sets does. Examples in section 7.5 show that some programs have partial stable models and no answer–set. Thus we say that *WFSX* generalizes answer–set semantics, in the sense that

[6]For nonpropositional programs, it was already shown (in section 7.5) that the computation of an answer–set cannot in general be made by finite approximations.

it assigns meaning to more programs.

For programs where both answer–set semantics and *WFSX* assign a meaning, the computational methods of the latter can be viewed as sound methods for the former:

Theorem 10.2.2 (Soundness wrt to answer–set semantics) *Let P be an extended logic program with at least one answer–set. Then* WFSX *is sound wrt the answer–set semantics, i.e. for every literal L :*

$$L \in \text{WFSX}(P) \quad \Rightarrow \quad L \in AS(P)$$

Proof: Follows directly from theorem 6.4.5, given the correspondence between Ω–extensions and PSMs (cf. theorem 6.6.1), and the correspondence between Reiter's extensions and answer–sets (cf. [Gelfond and Lifschitz, 1990]). $\diamondsuit$

This theorem only guarantees soundness for programs with answer-sets. As stated above in this section, the problem of determining whether a program has answer–sets is NP–complete. Thus, even though the methods of *WFSX* seem to be good sound computational methods for answer–sets, they are not as good for that purpose because one first has to determine the existence of answer–sets.

One way to define good computational methods for the decision problem in answer–set semantics is to restrict the class of programs (based on some syntatic criteria, in the spirit of [Dung, 1992b]) where those methods can be applied, and then use *WFSX*. The study of syntatic properties guaranteeing the existence of answer–sets and its equivalence to *WFSX*, i.e. guaranteeing that *WFSX* can be used to correctly compute answer–set semantics, is however beyond the scope of this work.

Bibliography

[Alferes and Pereira, 1992] J. J. Alferes and L. M. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 574–588. MIT Press, 1992.

[Alferes and Pereira, 1993a] J. J. Alferes and L. M. Pereira. Contradiction: when avoidance equal removal. Part I. In R. Dyckhoff, editor, *4th International Workshop on Extensions of Logic Programming*, pages 7–16. University of St. Andrews, 1993.

[Alferes and Pereira, 1993b] J. J. Alferes and L. M. Pereira. Logic programming, explicit negation, and autoepistemic logics. Technical report, AI Centre, Uninova, April 1993.

[Alferes *et al.*, 1993] J. J. Alferes, P. M. Dung, and L. M. Pereira. Scenario semantics of extended logic programs. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 334–348. MIT Press, 1993.

[Aparício, 1993] Joaquim Nunes Aparício. *Logic Programming: a tool for reasoning.* PhD thesis, Universidade Nova de Lisboa, 1993. To appear.

[Apt and Bezem, 1991] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[Apt and Bol, 1993] K. Apt and R. Bol. Logic programming and negation. Technical report, CWI and Eindhoven University of Technology, 1993. Draft.

[Apt *et al.*, 1988] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, 1988.

[Baral and Subrahmanian, 1990] C. Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics. In *International Workshop on Nonmonotonic Reasoning*, 1990.

[Baral and Subrahmanian, 1991] C. Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 69–86. MIT Press, 1991.

[Bidoit and Froidevaux, 1987] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription in stratified logic programming. In *Symposium on Principles of Database Systems*. ACM SIGACT-SIGMOD, 1987.

[Bidoit and Froidevaux, 1988] N. Bidoit and C. Froidevaux. General logic databases and programs: default logic semantics and stratification. *Journal of Information and Computation*, 1988.

[Birnbaum *et al.*, 1980] L. Birnbaum, M. Flowers, and R. McGuire. Towards an AI model of argumentation. In *Proceedings of AAAI'80*, pages 313–315. Morgan Kaufmann, 1980.

[Blair and Subrahmanian, 1987] H. Blair and V. S. Subrahmanian. Paraconsistent logic programming. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 340–360. Springer–Verlag, 1987.

[Bonatti, 1992] P. Bonatti. Autoepistemic logics as a unifying framework for the semantics of logic programs. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 417–430. MIT Press, 1992.

[Bonatti, 1993] P. Bonatti. Autoepistemic logic programming. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 151–167. MIT Press, 1993.

[Bondarenko *et al.*, 1993] A. Bondarenko, F. Toni, and R. Kowalski. An assumption–based framework for nonmonotonic reasoning. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 171–189. MIT Press, 1993.

[Brewka and Konolige, 1993] G. Brewka and K. Konolige. An abductive framework for generalized logic programs and other nonmonotonic systems. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[Brogi *et al.*, 1992] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. Normal logic programs as open positive programs. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 783–797. MIT Press, 1992.

[Bry, 1989] F. Bry. Logic programming as constructivism: a formalization and its applications to databases. In *Symposium on Priciples of Database Systems*, pages 34–50. ACM SIGACT–SIGMOD, 1989.

[Chandra and Harel, 1982] A. Chandra and D. Harel. Structure and complexity of relational queries. *JCSS*, 25(1):99–128, 1982.

[Chen and Warren, 1992] W. Chen and D. H. D. Warren. A goal–oriented approach to computing well–founded semantics. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 589–603. MIT Press, 1992.

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

[Cohen, 1987] R. Cohen. Analyzing the structure of argumentative discourse. *Computational Linguistics*, 13:11–24, 1987.

[Colmerauer *et al.*, 1973] A. Colmerauer, H. Kanoui, P. Russel, and R. Passero. Un systeme de communication homme–machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix–Marseille, 1973.

[Costa, 1974] N. Costa. On the theory of inconsistency formal system. *Notre Dame Journal of Formal Logic*, 15:497–510, 1974.

[Dix, 1991] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 166–180. MIT Press, 1991.

[Dix, 1992a] J. Dix. Classifying semantics of disjunctive logic programs. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 798–812. MIT Press, 1992.

[Dix, 1992b] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *3rd International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992.

[Dix, 1992c] J. Dix. A framework for representing and characterizing semantics of logic programs (extended version). Technical report, Institute for Logic, Complexity and Deduction Systems. University of Karlsruhe, December 1992.

[Dung and Ruamviboonsuk, 1991] P. M. Dung and P. Ruamviboonsuk. Well founded reasoning with classical negation. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 120–132. MIT Press, 1991.

[Dung *et al.*, 1992] P. M. Dung, A. C. Kakas, and P. Mancarella. Negation as failure revisited. Technical report, Asian Institute of Technology, University of Cyprus, and University of Pisa, 1992. Preliminary Report.

[Dung, 1991] P. M. Dung. Negation as hypotheses: An abductive framework for logic programming. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 3–17. MIT Press, 1991.

[Dung, 1992a] P. M. Dung. Logic programming as dialog–games. Technical report, Division of Computer Science, Asian Institute of Technology, December 1992.

[Dung, 1992b] P. M. Dung. On the relations between stable and well–founded models. *Theoretical Computer Science*, 105:7–25, 1992.

[Dung, 1993a] P. M. Dung. An argumentation semantics for logic programming with explicit negation. In D. S. Warren, editor, *10th International Conference on Logic Programming*, pages 616–630. MIT Press, 1993.

[Dung, 1993b] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[Emden and Kowalski, 1976] M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742, 1976.

[Eshghi and Kowalski, 1989] K. Eshghi and R. Kowalski. Abduction compared with negation by failure. In *6th International Conference on Logic Programming*. MIT Press, 1989.

[Etherington *et al.*, 1985] D. Etherington, R. Mercer, and R. Reiter. On the adequacy of predicate circumscription for closed–world reasoning. *Journal of Computational Intelligence*, 1:11–15, 1985.

[Fitting, 1985] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Gallaire *et al.*, 1984] H. Gallaire, J. Minker, and J. Nicolas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16:153–185, 1984.

[Geerts and Vermeir, 1993] P. Geerts and D. Vermeir. A nonmonotonic reasoning formalism using implicit specificity information. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 380–396. MIT Press, 1993.

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

[Gelder, 1989] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[Gelfond and Lifschitz, 1989] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In M. Reinfrank, J. de Kleer, M. Ginsberg, and E. Sandewall, editors, *Non–Monotonic Reasoning: 2nd International Workshop*, pages 74–99. LNAI 346, Springer–Verlag, 1989.

[Gelfond and Lifschitz, 1990] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

[Gelfond and Lifschitz, 1992] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *International Joint Conference and Symposium on Logic Programming*, pages 559–573. MIT Press, 1992.

[Gelfond *et al.*, 1989] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38:75–94, 1989.

[Gelfond, 1987] M. Gelfond. On stratified autoepistemic theories. In *AAAI'87*, pages 207–211. Morgan Kaufmann, 1987.

[Gurevich and Shelah, 1986] Y. Gurevich and S. Shelah. Fixed–point extensions of first order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[Hintikka, 1983] J. Hintikka. *The Game of Language*. Reidel Publishing Company, 1983.

[HMSO, 1981] HMSO. *British Nationality Act*. Her Majesty's Stationery Office, 1981.

[Immerman, 1986] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1):86–104, 1986.

[Inoue, 1991] K. Inoue. Extended logic programs with default assumptions. In Koichi Furukawa, editor, *8th International Conference on Logic Programming*, pages 490–504. MIT Press, 1991.

[Kakas and Mancarella, 1991a] A. C. Kakas and P. Mancarella. Negation as stable hypothesis. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 275–288. MIT Press, 1991.

[Kakas and Mancarella, 1991b] A. C. Kakas and P. Mancarella. Stable theories for logic programs. In Ueda and Saraswat, editors, *International Logic Programming Symposium*, pages 85–100. MIT Press, 1991.

[Kakas *et al.*, 1993] A. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1993.

[Kautz and Selman, 1989] H. A. Kautz and B. Selman. Hard problems for simple default logics. In R. Brachman, H. Levesque, and R. Reiter, editors, *1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 189–197. Morgan Kaufmann, 1989.

[Kautz, 1986] H. Kautz. The logic of persistence. In *AAAI'86*, pages 401–405. Morgan Kaufmann, 1986.

[Kifer and Lozinskii, 1989] M. Kifer and E. L. Lozinskii. A logic for reasoning with inconsistency. In *4th IEEE Symposium on Logic in Computer Science*, pages 253–262, 1989.

[Konolige, 1982] K. Konolige. Circumscriptive ignorance. In *AAAI'82*, pages 202–204, 1982.

[Konolige, 1992] K. Konolige. Using default and causal reasoning in diagnosis. In B. Nebel, C. Rich, and W. Swartout, editors, *3rd International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992.

[Kowalski and Sadri, 1990] R. Kowalski and F. Sadri. Logic programs with exceptions. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*. MIT Press, 1990.

[Kowalski, 1974] R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP'74*, pages 569–574, 1974.

[Kowalski, 1979] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

[Kowalski, 1989] R. Kowalski. The treatment of negation in logic programs for representing legislation. In *2nd International Conference on Artificial Intelligence and Law*, pages 11–15, 1989.

[Kowalski, 1990] R. Kowalski. Problems and promises of computational logic. In John Lloyd, editor, *Computational Logic*, pages 1–36. Basic Research Series, Springer–Verlag, 1990.

[Kowalski, 1991] R. Kowalski. Legislation as logic programs. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[Kraus *et al.*, 1990] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.

[Kunen, 1987] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Kunen, 1988] K. Kunen. Some remarks on the completed database. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 978–992. MIT Press, 1988.

[Levesque, 1986] H. Levesque. Making believers out of computers. *Artificial Intelligence*, 30:81–107, 1986.

[Lifschitz, 1985] V. Lifschitz. Computing circumscription. In *International Joint Conference on Artificial Intelligence*, pages 121–127. Morgan Kaufmann, 1985.

[Lifschitz, 1991a] V. Lifschitz. Logic and actions. In *5th Portuguese Artificial Intelligence Conference*, 1991. Invited talk.

[Lifschitz, 1991b] V. Lifschitz. Minimal belief and negation as failure. Technical report, Department of Computer Science and Department of Philisophy, University of Texas at Austin, 1991.

[Lifschitz, 1991c] V. Lifschitz. Nonmonotonic databases and epistemic queries. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1991.

[Lin and Shoham, 1990] F. Lin and Y. Shoham. Epistemic semanitcs for fixed–points non-monotonic logics. In R. Parikh, editor, *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the Third Conference*, pages 111–120, 1990.

[Lin, 1988] F. Lin. Circumscription in a modal logic. In M. Vardi, editor, *2nd International Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 113–127, 1988.

[Lloyd and Topor, 1985] J. Lloyd and R. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2:93–109, 1985.

[Lloyd and Topor, 1986] J. Lloyd and R. Topor. A basis for deductive database systems II. *Journal of Logic Programming*, 3:55–67, 1986.

[Lloyd, 1984] J. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 1984.

[Lloyd, 1987] J. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2), 1987.

[Marek and Truszczynski, 1991] W. Marek and M. Truszczynski. Autoepistemic logics. *Journal of the ACM*, 38(3):588–619, 1991.

[McCarthy, 1980] J. McCarthy. Circumscription - a form of non–monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McCarthy, 1986] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26:89–116, 1986.

[McDermott, 1982] D. McDermott. Non–monotonic logic II. *Journal of the ACM*, 29(1):33–57, 1982.

[McGuire *et al.*, 1981] R. McGuire, L. Birnbaum, and M. Flowers. Towards an AI model of argumentation. In *International Joint Conference on Artificial Intelligence*, pages 58–60. Morgan Kaufmann, 1981.

[Minker, 1987] J. Minker. On indefinite databases and the closed world assumption. In M. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 326–333. Morgan Kaufmann, 1987.

[Minker, 1988] J. Minker. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[Monteiro, 1992] L. Monteiro. Notes on the negation in logic programs. Technical report, Department of Computer Science, Univerdade Nova de Lisboa, 1992. Course Notes, 3rd Advanced School on Artificial Intelligence, Azores, Portugal, 1992.

[Moore, 1985] R. Moore. Semantics considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Morris, 1988] P. H. Morris. Autoepistemic stable clousure and contradiction resolution. In *2nd Workshop on Nonmonotonic Reasoning*, pages 60–73, 1988.

[Nejdl *et al.*, 1993] W. Nejdl, G. Brewka, L. Consolle, P. Mancarella, and L. M. Pereira. *LAP – Logic Agents Programming*. ESPRIT BRA proposal (no. 8099), April 1993.

[Nelson, 1949] D. Nelson. Constructible falsity. *JSL*, 14:16–26, 1949.

[Nerode *et al.*, 1991] A. Nerode, W. Marek, and V. S. Subrahmanian, editors. *Logic Programming and Non–monotonic Reasoning: Proceedings of the First International Workshop*, Washington D.C., USA, 1991. The MIT Press.

[Nute, 1986] D. Nute. Ldr : A logic for defeasible reasoning. Technical report, Advanced Computational Center, University of Georgia, 1986.

[Pearce and Wagner, 1990] D. Pearce and G. Wagner. Reasoning with negative information I: Strong negation in logic programs. In L. Haaparanta, M. Kusch, and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*, pages 430–453. Acta Philosophica Fennica 49, 1990.

[Pearce, 1992a] D. Pearce. A cumulative extension of constructive default logic. Technical report, Gruppe Logik, Wissenstheorie & Information, Institut fur Philosophie, Freie Universitat Berlin, 1992. Preliminary Version.

[Pearce, 1992b] D. Pearce. Default logic and constructive logic. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 309–313. John Wiley & Sons, 1992.

[Pereira and Alferes, 1992] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conference on Artificial Intelligence*, pages 102–106. John Wiley & Sons, 1992.

[Pereira and Alferes, 1993a] L. M. Pereira and J. J. Alferes. Contradiction: when avoidance equal removal. Part II. In R. Dyckhoff, editor, *4th International Workshop on Extensions of Logic Programming*, pages 17–26. University of St. Andrews, 1993.

[Pereira and Alferes, 1993b] L. M. Pereira and J. J. Alferes. Optative reasoning with scenario semantics. In D. S. Warren, editor, *10th International Conference on Logic Programming*, pages 601–615. MIT Press, 1993.

[Pereira and Calejo, 1988] L. M. Pereira and M. Calejo. A framework for Prolog debugging. In R. Kowalski, editor, *5th International Conference on Logic Programming*. MIT Press, 1988.

[Pereira and Nerode, 1993] L. M. Pereira and A. Nerode, editors. *Logic Programming and Non–monotonic Reasoning: Proceedings of the Second International Workshop*, Lisboa, Portugal, 1993. The MIT Press.

[Pereira *et al.*, 1991a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 105–119. MIT Press, 1991.

[Pereira *et al.*, 1991b] L. M. Pereira, J. J. Alferes, and J. N. Aparício. The extended stable models of contradiction removal semantics. In P. Barahona, L. M. Pereira, and A. Porto, editors, *5th Portuguese Artificial Intelligence Conference*, pages 105–119. LNAI 541, Springer–Verlag, 1991.

[Pereira *et al.*, 1991c] L. M. Pereira, J. J. Alferes, and J. N. Aparício. A practical introduction to well founded semantics. In B. Mayoh, editor, *Scandinavian Conference on Artificial Intelligence*. IOS Press, 1991.

[Pereira *et al.*, 1991d] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Counterfactual reasoning based on revising assumptions. In Ueda and Saraswat, editors, *International Logic Programming Symposium*, pages 566–577. MIT Press, 1991.

[Pereira *et al.*, 1991e] L. M. Pereira, J. N. Aparício, and J. J. Alferes. A derivation procedure for extended stable models. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1991.

[Pereira *et al.*, 1991f] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Hypothetical reasoning with well founded semantics. In B. Mayoh, editor, *Scandinavian Conference on Artificial Intelligence*. IOS Press, 1991.

[Pereira *et al.*, 1991g] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In Koichi Furukawa, editor, *8th International Conference on Logic Programming*, pages 475–489. MIT Press, 1991.

[Pereira *et al.*, 1992a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Adding closed world assumptions to well founded semantics. In *Fifth Generation Computer Systems*, pages 562–569. ICOT, 1992.

[Pereira *et al.*, 1992b] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction removal semantics with explicit negation. In *Applied Logic Conference*. Preproceedings by ILLC, Amsterdam, 1992. To appear in Springer–Verlag LNAI.

[Pereira *et al.*, 1992c] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Default theory for well founded semantics with explicit negation. In D. Pearce and G. Wagner, editors, *Logics in AI. Proceedings of the European Workshop JELIA'92*, pages 339–356. LNAI 633, Springer–Verlag, 1992.

[Pereira *et al.*, 1992d] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Well founded semantics with explicit negation and default theory. Technical report, AI Centre, Uninova, March 1992. Submitted to the Journal of Logic Programming.

[Pereira *et al.*, 1992e] L. M. Pereira, J. J. Alferes, and C. Damásio. The sidetracking meta principle. In *Simpósio Brasileiro de Inteligência Artificial*, pages 229–242, 1992.

[Pereira *et al.*, 1992f] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Logic programming for nonmonotonic reasoning. In *Applied Logic Conference*. Preproceedings by ILLC, Amsterdam, 1992. To appear in Springer–Verlag LNAI.

[Pereira *et al.*, 1993a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Adding closed world assumptions to well founded semantics (extended improved version). *Theoretical Computer Science. Special issue on selected papers from FGCS'92*, 1454, 1993.

[Pereira *et al.*, 1993b] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of Logic Programming. Special issue on Nonmonotonic reasoning*, 1993. To appear.

[Pereira *et al.*, 1993c] L. M. Pereira, C. Damásio, and J. J. Alferes. Debugging by diagnosing assumptions. In *1st International Workshop on Automatic Algorithmic Debugging, AADEBUG'93*. Preproceedings by Linkoping University, 1993. To appear in Springer–Verlag LNCS.

[Pereira *et al.*, 1993d] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 316–330. MIT Press, 1993.

[Pereira *et al.*, 1993e] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal in logic programs. In L. Damas and M. Filgueiras, editors, *6th Portuguese Artificial Intelligence Conference*. Springer–Verlag, 1993. To appear.

[Pollock, 1992] J. L. Pollock. How to reason defeasibly. *Artificial Intelligence*, 57:1–42, 1992.

[Poole, 1988] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[Przymusinska and Przymusinski, 1988] H. Przymusinska and T. Przymusinski. Weakly perfect model semantics. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1106–1122. MIT Press, 1988.

[Przymusinska and Przymusinski, 1990] 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.

[Przymusinska and Przymusinski, 1991] H. Przymusinska and T. Przymusinski. Nonmonotonic reasoning and logic programming - Advanced Tutorial. Technical report, Department of Computer Science, California State Polytechnic and Department of Computer Science, University of California at Riverside, 1991.

[Przymusinska and Przymusinski, 1993] H. Przymusinska and T. Przymusinski. Stationary default extensions. Technical report, Department of Computer Science, California State Polytechnic and Department of Computer Science, University of California at Riverside, 1993.

[Przymusinski, 1988] T. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[Przymusinski, 1989a] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *8th Symposium on Principles of Database Systems*. ACM SIGACT-SIGMOD, 1989.

[Przymusinski, 1989b] T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Przymusinski, 1989c] T. Przymusinski. Three–valued non–monotonic formalisms and logic programming. In R. Brachman, H. Levesque, and R. Reiter, editors, *1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 341–348. Morgan Kaufmann, 1989.

[Przymusinski, 1990a] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 459–477. MIT Press, 1990.

[Przymusinski, 1990b] T. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In Debray and Hermenegildo, editors, *North American Conference on Logic Programming*, pages 40–57. MIT Press, 1990.

[Przymusinski, 1991a] T. Przymusinski. Autoepistemic logic of closed beliefs and logic programming. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and NonMonotonic Reasoning*, pages 3–20. MIT Press, 1991.

[Przymusinski, 1991b] T. Przymusinski. A semantics for disjunctive logic programs. In Loveland, Lobo, and Rajasekar, editors, *ILPS'91 Workshop in Disjunctive Logic Programs*, 1991.

[Rajasekar *et al.*, 1989] A. Rajasekar, J. Lobo, and J. Minker. Weak generalized closed world assumptions. *Automated Reasoning*, 5:293–307, 1989.

[Reiter, 1978] R. Reiter. On closed–world data bases. In H. Gallaire and J. Minker, editors, *Logic and DataBases*, pages 55–76. Plenum Press, 1978.

[Reiter, 1980] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:68–93, 1980.

[Reiter, 1984] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie and J. Mylopoulos, editors, *On Conceptual Modelling*, pages 191–233. Springer–Verlag, 1984.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–96, 1987.

[Reiter, 1990] R. Reiter. On asking what a database knows. In John Lloyd, editor, *Computational Logic*, pages 96–113. Basic Research Series, Springer–Verlag, 1990.

[Ross and Topor, 1988] K. Ross and R. Topor. Inferring negative information from disjunctive databases. *Automated Reasoning*, 4:397–424, 1988.

[Sakama, 1992] C. Sakama. Extended well–founded semantics for paraconsistent logic programs. In *Fifth Generation Computer Systems*, pages 592–599. ICOT, 1992.

[Shanahan, 1992] M. Shanahan. Explanations in the situation calculus. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, 1992.

[Shepherdson, 1988] J. Shepherdson. Negation in logic programming for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[Shepherdson, 1990] J. Shepherdson. Negation as failure, completion and stratification. In *Handbook of Artificial Intelligence and Logic Programming*, 1990.

[Shoham, 1986] Y. Shoham. Chronological ignorance: Time, nonmonotonicity, necessity and causal theories. In *American Association for Artificial Intelligence*, pages 389–393, 1986.

[Stein, 1989] L. J. Stein. Skeptical inheritance: computing the intersection of credulous extensions. In *International Joint Conference on Artificial Intelligence*, pages 1153–1158. Morgan Kaufmann Publishers, 1989.

[Tarski, 1955] A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Toulmin, 1958] S. Toulmin. *The uses of arguments*. Cambridge University Press, 1958.

[Touretzky *et al.*, 1987] D. S. Touretzky, J. F. Horty, and R. H. Thomason. A clash of intuitions: the current state of nonmonotonic multiple inheritance systems. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.

[Vardi, 1982] M. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–145, 1982.

[Wagner, 1991a] G. Wagner. A database needs two kinds of negation. In B. Thalheim, J. Demetrovics, and H-D. Gerhardt, editors, *Mathematical Foundations of Database Systems*, pages 357–371. LNCS 495, Springer–Verlag, 1991.

[Wagner, 1991b] G. Wagner. Ex contradictione nihil sequitur. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1991.

[Wagner, 1993] G. Wagner. Reasoning with inconsistency in extended deductive databases. In L. M. Pereira and A. Nerode, editors, *2nd International Workshop on Logic Programming and NonMonotonic Reasoning*, pages 300–315. MIT Press, 1993.

[Warren *et al.*, 1977] D. H. Warren, L. M. Pereira, and F. Pereira. Prolog: The language and its implementation compared with Lisp. In *Symposium on Artificial Intelligence and Programming Languages*, pages 109–115. ACM SIGPLAN–SIGART, 1977.

[Warren, 1989] D.S. Warren. The XWAM: A machine that integrates Prolog and deductive databases. Technical report, SUNY at Stony Brook, 1989.

Part III

Appendices

Appendix A

A Prolog top–down interpreter for WFSX

Here, for the sake of completeness, we present a Prolog top–down interpreter for *WFSX*.

This interpreter is based on the derivations procedures for well–founded semantics of normal logic program of [Pereira *et al.*, 1991e], and results from of generalization of it as suggested in page 37.

The code of the interpreter follows closely the code of an interpreter for WFS, described in [Pereira *et al.*, 1992e]. Its correctness for propositional programs follows directly from the results of [Aparício, 1993] regarding derivation procedures for *WFSX*.

For this interpreter, programs are sets of rules of the form:

```
H :- B1, ..., Bn, not C1, ..., not Cm
```

where `H`, `B1`, ..., `Bn`, `C1`, ..., and `Cm` are predicates or terms of the form `-P` where `P` is a predicate. `-P` stands for the explicit negation of `P`.

The goal `demo( G )` succeeds if the literal `G` is true in the well–founded model of the program, and fails otherwise.

```
% ************************************************
%                Meta Level demo predicate
% ************************************************

demo( G ) :- demo( G, [[]] ).

demo( not true, _ ) :- !, fail.
demo( true, _ ).
demo( G, Cx ) :-
        prunning( G, Cx, Com ), !, Com.
demo( (A,B), Cx ) :-
        demo( A, Cx ), demo( B, Cx ).
demo( G, Cx ) :-
        add_to_Cx( G, Cx, NCx ),
        rule( G, Body ),
        demo( Body, NCx ).
```

```
% ************************************************
%            Determining rules for literals
% ************************************************

rule( not G, B ) :-
        clause_neg( G, B ).
rule( G, B ) :-
        clause( G, B ).

% Rules for default literal 'not G'
clause_neg( G, true ) :-                    % a fact if G has no rules
        not clause( G, _ ).
clause_neg( G, Body ) :-
        findall( B, clause(G,B), L ),
        one_from_each( L, Body ).
clause_neg( G, B ) :-                       % The coherence principle
        obj_compl(G,ObjCG),
        clause( ObjCG, B ).

one_from_each( [B1], CSB ) :- !,
        member_conj( B1, SB ),
        compl( SB, CSB ).
one_from_each( [B1|RestBodies], (CSB,SRest) ) :-
        member_conj( B1, SB ),
        compl( SB, CSB ),
        one_from_each( RestBodies, SRest ).

% Selects, in turn, each literal from a clause
member_conj( A, A ) :-
        A \= ( _, _ ).
member_conj( (A,_), A ).
member_conj( (_,B), A) :-
        member_conj( B ,A ).

% Default complement
compl( not G, G ) :- !.
compl( G, not G ).

% Objective complement
obj_compl( -G, G ) :- !.
obj_compl( G, -G ).


% ************************************************
%                       Pruning
% ************************************************

% loop in 'not G' in the last context succeds
prunning( not G, [LCx|_], true ) :-
        member( not G, LCx).
```

```
% loop detection in different contexts
prunning( G, Cx, fail ) :-
        in_all_Cx( G, Cx ).

% Detects if either G or its complement is in the contexts
in_all_Cx( G, [[G|_]|_] ).
in_all_Cx( G, [[NG|_]|_] ) :- compl( G, NG ).

in_all_Cx( G, [[]|Cx] ) :-
        in_all_Cx( G, Cx ).
in_all_Cx( G, [[_|T]|Cx] ) :-
        in_all_Cx( G, [T|Cx] ).

% ************************************************
%                Context Management
% ************************************************

add_to_Cx( G, [[]], [[G]] ) :- !.
add_to_Cx( G, [Sn|OtherS], [[G|Sn]|OtherS] ) :-
        same_sign( G, Sn ), !.
add_to_Cx( G, [Sn|OtherS], [[G]|[Sn|OtherS]] ).

% same_sign decides if a literal has the same (default) sign
% of the current context
same_sign( not _, [not _|_] ).
same_sign( G, [C|_] ) :-
        G \= (not _),
        C \= (not _).
```

Appendix B

Additional definitions

In this section we recall the definition and some properties of support sets of normal logic programs, introduced in [Pereira *et al.*, 1991a].

Definition B.0.1 (Support set) *A Support Set of a literal L belonging to the WF Model M_P of a program P, represented as $SS_P(L)$, or $SS(L)$ for short, is obtained as follows:*

- *If L is an atom:*
  - *Choose some rule of P for L where all the literals in its body belong to M_P. One $SS(L)$ is obtained by taking all those body literals plus the literals in some SS of each body literal.*
- *If $L = not\ A$:*
  - *If there are no rules for A in P then the only $SS(L)$ is $\{\}$.*
  - *Otherwise, choose from each rule defined for A, a literal such that its complement belongs to M_P. A $SS(L)$ has all those complement literals, and the literals of a SS of each of them.*

By considering all possible rules of P for a literal all its SSs are obtained.

Here we define $Rules(SS_P(L)) \subseteq P$ as the rules used in the definition above to build $SS_P(L)$.

Proposition B.0.1 (Existence of support set) *Every literal L belonging to the well–founded model of a program P has at least one support set $SS_P(L)$.*

Since by definition every literal L with a support set $SS_P(L)$ belong to the WFM of P, we can say that a literal has at least one support set iff it belongs to the WFM.

Other properties of support sets, which are used in some proofs of this paper, are presented below.

Proposition B.0.2 *For any atom A such that $A \in WFM(P)$, there is at least one support set S of A such that $A \notin S$ and not $A \notin S$.*

Proposition B.0.3 *Let P be a program, $A \in WFM(P)$ be an atom, and $SS_P(A)$ a support set of A. Then $A \in WFM(P')$ for every program P' such that:*

$$Rules(SS_P(A)) \subseteq P' \subseteq P$$

Appendix C

Proofs of theorems

Proof of theorem 5.1.1: We prove this theorem here only for the case of a stationary semantics. The proof for stable semantics is quite similar and is omitted.

$\Rightarrow$ If a stationary semantics is coherent then for any P^* every model M of P^* having $\neg\_A$ also has not_A. By proposition 5.1.1:

$$not\_A \in M \Leftrightarrow \sim A \in M.$$

Similarly we conclude that for every M :

$$A \in M \text{ iff } \sim\neg\_A \in M.$$

Thus, given that models of clausal programs are always total, every model containing A does not contains $\neg\_A$, and every model containing $\neg\_A$ does not contain A, which is the consistency requirement.

$\Leftarrow$ If a stationary semantics is consistent then for any P^* every model M of P^* having $\neg\_A$ does not have A, and vice–versa. By proposition 5.1.1:

$$A \notin M \Leftrightarrow \sim A \in M \Leftrightarrow not\_A \in M.$$

Similarly we conclude that for every M :

$$\neg\_A \notin M \Leftrightarrow \sim\neg\_A \in M \Leftrightarrow not\_\neg\_A \in M.$$

Thus:

$$\neg\_A \in M \Rightarrow A \notin M \Rightarrow not\_A \in M$$

and

$$A \in M \Rightarrow \neg\_A \notin M \Rightarrow not\_\neg\_A \in M$$

which, by definition of $AX_\neg$ model, is equivalent to coherence.

Proof of theorem 5.1.4: Consider the fixpoint equation:

$$P^* = \neg\_P \cup AX_\neg \cup \Big\{not\_L \mid P^* \models_{\overline{CIRC}} \sim L\Big\} \cup \Big\{\sim not\_L \mid P^* \models_{\overline{CIRC}} L\Big\}.$$

By definition, the expansions of the stationary semantics with classical negation are the fixpoints of the equation obtained from the one above by deleting the set of axioms $AX_\neg$ and

replacing in $\neg\_P$ every occurence of an objective literal $\neg\_L$ by $\sim L$. Hereafter we denote such programs by $\sim\_P$.

Let $P_1^* = \neg\_P \cup AX_\neg \cup S$ be a stationary $AX_\neg$ expansion of P, and let $P_2^* = \sim\_P \cup S$. We prove that P_2^* is an expansion of the stationary semantics with classical negation.

For every objective proposition L, by the axioms in $AX_\neg$, $\neg\_L \Leftrightarrow \sim L$. So, it is clear that the models of P_1^* are the models of P_2^* modulo propositions of the form $\neg\_L$. Thus for every atom A :

$$P_1^* \models A \Leftrightarrow P_2^* \models A \qquad (+)$$

We now prove that for every atom A:

$$P_1^* \models\!\!\!\!\!\!\sim_{CIRC} \sim A \Leftrightarrow P_2^* \models\!\!\!\!\!\!\sim_{CIRC} \sim A \qquad (\&)$$

($\Rightarrow$) Let $M_1', \ldots, M_n'$ be all the minimal models of P_1^*, and let M_i'' be the model obtained from M_i' by removing all propositions of the form $\neg\_L$.

As we've seen above, all such M_i'' are models of P_2^* and, as only positive propositions are removed, they are also the minimal models of P_2^*. Thus, if $\sim A$ is a consequence of all minimal models of P_1^* it is also a consequence of all minimal models of P_2^*.

($\Leftarrow$) Let $M_1'', \ldots, M_n''$ be all the minimal models of P_2^*, and let

$$M_i' = M_i'' \cup \{\neg\_L \mid L \notin M_i''\}.$$

All such M_i' are models of P_1^*.

Let us assume that one M_i' is not a minimal models of P_1^*, i.e. there exists a model N of P_1^* such that $N \bar{\leq} M_i'$, and $N \neq M$. In such a case, by definition of $\bar{\leq}$:

$$N_{pos} \subseteq M_i'' \vee (N_{pos} = M_i'' \wedge N \subseteq M_i')$$

where N_{pos} is the subset of N obtained by deleting from it all literals of the form $\neg\_L$.

Clearly N_{pos} is a model of P_2^*. Thus, if the first disjunct holds, M_i'' is not a minimal model of P_2^*, which contradicts one of our hypotheses.

If $N_{pos} = M_i''$ then for N to be a model of P_1^*, by the axioms in $AX_\neg$, for every atom A :

$$A \notin N_{pos} \Rightarrow \neg\_A \in N$$

So, by definition of M_i'' :

$$M_i'' \supseteq N$$

which also contradicts our hypotheses.

With the results above we now finalize the proof that P_2^* is an expansion. Since P_1^* is an expansion:

$$S = \left\{not\_L \mid P_1^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L\right\} \cup \left\{\sim not\_L \mid P_1^* \models\!\!\!\!\!\!\sim_{CIRC} L\right\}$$

By (&) :

$$P_1^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L \Leftrightarrow P_1^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L$$

As already mentioned in page 43, it is known (cf. [Lifschitz, 1985, Etherington *et al.*, 1985, Gelfond *et al.*, 1989]) that for any proposition A of any theory T :

$$T \models\!\!\!\!\!\!\sim_{CIRC} A \equiv T \models A$$

Thus:

$$P_1^* \models\!\!\!\models_{CIRC} L \Leftrightarrow P_1^* \models L \overset{\text{by}\,(+)}{\Leftrightarrow} P_2^* \models L \Leftrightarrow P_2^* \models\!\!\!\models_{CIRC} L$$

Replacing in S these equivalence results:

$$S = \left\{not\_L \mid P_2^* \models\!\!\!\models_{CIRC} \sim L\right\} \cup \left\{\sim not\_L \mid P_2^* \models\!\!\!\models_{CIRC} L\right\}$$

Recall that by definition $P_2^* = \sim\_P \cup S$ so, replacing S by its value:

$$P_2^* = \sim\_P \cup \left\{not\_L \mid P_2^* \models\!\!\!\models_{CIRC} \sim L\right\} \cup \left\{\sim not\_L \mid P_2^* \models\!\!\!\models_{CIRC} L\right\}$$

i.e. P_2^* is an expansion of the stationary semantics with classical negation.

The proof that every expansion of the stationary semantics with classical negation corresponds to a stationary $AX_\neg$ expansion is similar to the one above, and is omitted.

Proof of theorem 5.1.6:

Proving the equivalence between the two alternative definitions is trivial. Thus we only prove the equivalence between *WFSX* and the second definition presented in the theorem.

Without loss of generality (cf. theorem 10.1.2) we assume that programs are in the semantic kernel form, i.e. a program is a set of rules of the form:

$$L \leftarrow not\ A_1, \ldots, not\ A_n \quad n \geq 0$$

We begin by proving a lemma:

Lemma C.0.3 *Let $\neg\_P$ be a clausal program and let P^+ be:*

$$P^+ = \neg\_P \cup S_n \cup S_p$$

where S_n is a set of default literals of the form not_L, and S_p is a set of default literals of the form $\sim not\_L$.

For every clause with L in $\neg\_P$, i.e. of the form:

$$L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P$$

there exists $\sim not\_A_i \in S_p$, iff

$$P^+ \models\!\!\!\models_{CIRC} \sim L$$

Proof:

($\Rightarrow$) Let $\sim not\_A_j$ be one literal in the j-th clause with L such that $\sim not\_A_j \in S_p$[1].

Then all models of P^+ contain a set of such $\sim not\_A_j$:

$$\{\sim not\_A_1, \ldots, \sim not\_A_m\}$$

where m is the number of clauses with literal L.

Thus every clause with L is satisfied by all models of P^+ independently of the truth value of L, and thus:

$$P^+ \models\!\!\!\models_{CIRC} \sim L$$

[1] By hypothesis such a literal always exists.

($\Leftarrow$) Assume the contrary, i.e. there exists a clause:

$$L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P$$

such that

$$\{\sim not\_A_1, \ldots, \sim not\_A_n\} \cap S_p = \{\}$$

and $P^+ \models\!\!\!\!\!\!\sim_{CIRC} \sim L$.

Then, given that, by the form of programs, literals of the form $\sim not\ L$ can only be a consequence of P^+ if they belong to S_p, there exists a model M of the circumscription such that

$$\{not\_A_1, \ldots, not\_A_n\} \subseteq M$$

and thus L belongs to that model. So $P^+ \not\models\!\!\!\!\!\!\sim_{CIRC} \sim L$.

$\diamondsuit$

Given that complete scenaria (definition 7.3.1 in section 7) correspond to partial stable models (cf. theorem 7.4.9), it is enough to prove that there is a one to one correspondence between the fixpoints of:

$$P^* = \neg\_P \cup \left\{not\_L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L \text{ or } P^* \models \neg\_L\right\} \cup \{\sim not\ L \mid P^* \models L\}^2$$

and complete scenaria.

This correspondence is proven in two parts:

- first we prove that if $P \cup H$ is a complete scenario then

  $$P^* = \neg\_P \cup H \cup \{\sim not\_L \mid P \cup H \vdash L\}$$

  is an expansion.

- then we prove that if P^* is an expansion then

  $$P \cup \left\{not\ L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L \text{ or } P^* \models \neg\_L\right\}$$

  is a complete scenario.

Let us assume that $P \cup H$ is a complete scenario. i.e.

$$\begin{array}{ll} (i) & not\ L \in H \Rightarrow not\ L \in Mand(H) \text{ or } not\ L \in Acc(H) \\ (ii) & not\ L \in Mand(H) \Rightarrow not\ L \in H \\ (iii) & not\ L \in Acc(H) \Rightarrow not\ L \in H \end{array}$$

We show that

$$P^* = \neg\_P \cup H \cup \{\sim not\_L \mid P \cup H \vdash L\}$$

is an expansion of the clausal program $\neg\_P$ of P, i.e.

$$H \cup \{\sim not\_L \mid P \cup H \vdash L\} = \left\{not\_L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L \text{ or } P^* \models \neg\_L\right\} \cup \{\sim not\ L \mid P^* \models L\}$$

We'll do that by separately proving the two equalities:

$$\{\sim not\_L \mid P \cup H \vdash L\} = \{\sim not\ L \mid P^* \models L\} \qquad (eq1)$$
$$H = \left\{not\_L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L \text{ or } P^* \models \neg\_L\right\} \qquad (eq2)$$

[2] Within this proof we designate expansions as such fixpoints

To prove the first equality we have to show that

$$P \cup H \vdash L \Leftrightarrow P^* \models L$$

By definition of $\vdash$, $P \cup H \vdash L$ iff there exists a rule

$$L \leftarrow not\ A_1, \ldots, not\ A_n \in P$$

such that

$$\{not\ A_1, \ldots, not\ A_n\} \subseteq H$$

By definition of clausal program $\neg\_P$ of a program P, such a rule exists iff

$$L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P$$

And, by construction of P^*,

$$\{not\ A_1, \ldots, not\ A_n\} \subseteq H \ \Leftrightarrow\ \{not\ A_1, \ldots, not\ A_n\} \subseteq P^*$$

Thus, clearly $P^* \models L$.

For the equality $(eq2)$, we have to prove that:

1. $P^* \models \neg\_L \Rightarrow not\ L \in H$
2. $P^* \models_{\overline{CIRC}} \sim L \Rightarrow not\ L \in H$
3. $not\ L \in H \Rightarrow P^* \models_{\overline{CIRC}} \sim L\ or\ P^* \models \neg\_L$

1. By equality $(eq1)$:
$$P^* \models \neg\_L \Leftrightarrow P \cup H \vdash \neg L$$
and by definition of $Mand(H)$:
$$P \cup H \vdash \neg L \Rightarrow not\ L \in Mand(H) \overset{\text{by (ii)}}{\Rightarrow} not\ L \in H$$

2. By lemma C.0.3, $P^* \models_{\overline{CIRC}} \sim L$ iff
$$\forall L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P \mid \exists \sim not\_A_i \in P^*$$
By construction of P^*, $\sim not\_A_i \in P^* \Leftrightarrow P \cup H \vdash A_i$.
By definition of clausal program $\neg\_P$ of a program P :
$$\begin{array}{c} L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P \\ \Leftrightarrow \\ L \leftarrow not\ A_1, \ldots, not\ A_n \in P \end{array}$$
By definition of acceptable hypotheses, if for every rule
$$L \leftarrow not\ A_1, \ldots, not\ A_n \in P$$
there exists an A_i such that $P \cup H \vdash A_i$, then $not\ L \in Acc(H)$.
Thus:
$$P^* \models_{\overline{CIRC}} \sim L \Rightarrow not\ L \in Acc(H) \overset{\text{by (iii)}}{\Rightarrow} not\ L \in H$$

3. By (i) :

$$not\ L \in H \Rightarrow not\ L \in Mand(H)\ or\ not\ L \in Acc(H)$$

If $not\ L \in Mand(H)$ then, by definition of $Mand(H)$:

$$P \cup H \vdash \neg L$$

and, by equality $(eq1)$, $P^* \models \neg\_L$.

If $not\ L \in Acc(H)$ then, by definition of $Acc(H)$, for every rule of the form

$$L \leftarrow not\ A_1, \ldots, not\ A_n \in P$$

there exists an A_i such that $P \cup H \vdash A_i$.
By construction of P^*, if $P \cup H \vdash A_i$ then $\sim not\_A_i \in P^*$. Thus, for every clause

$$L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n \in \neg\_P$$

there exists $\sim not\_A_i \in P^*$, and by lemma C.0.3, $P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L$.

- Let us assume that P^* is an expansion, i.e.

$$P^* = \neg\_P \cup \left\{not\_L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L\ or\ P^* \models \neg\_L\right\} \cup \{\sim not\ L \mid P^* \models L\}$$

We prove now that

$$P \cup \left\{not\ L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L\ or\ P^* \models \neg\_L\right\}$$

is a complete scenario, i.e. by making

$$H = \left\{not\ L \mid P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L\ or\ P^* \models \neg\_L\right\}$$

the above conditions (i), (ii), and (iii) hold.

(i) By definition of H :

$$not\ L \in H \Rightarrow P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L\ or\ P^* \models \neg\_L$$

Similarly to the proof in point 2 above, it is easy to prove that if $P^* \models\!\!\!\!\!\!\sim_{CIRC} \sim L$ then $not\ L \in Acc(H)$, and that if $P^* \models \neg\_L$ then $not\ L \in Mand(H)$. So:

$$not\ L \in H \Rightarrow not\ L \in Mand(H)\ or\ not\ L \in Acc(H)$$

(ii) By definition of $Mand(H)$:

$$not\ L \in Mand(H) \Rightarrow P \cup H \vdash \neg L$$

Thus there exists a rule in P of the form

$$\neg L \leftarrow not\ A_1, \ldots, not\ A_n$$

such that

$$\{not\ A_1, \ldots, not\ A_n\} \subseteq H$$

So, by definition of H and given that P^* is an expansion, there is a clause in $\neg\_P$ of the form

$$\neg\_L \vee \sim not\_A_1 \vee \ldots \vee \sim not\_A_n$$

such that

$$\{not\_A_1, \ldots, not\_A_n\} \subseteq P^*$$

and clearly $P^* \models \neg\_L$. Thus, by definition of H, $not\ L \in H$.

(*iii*) If $not\ L \in Acc(H)$ then, by definition of $Acc(H)$, for every rule of the form

$$L \leftarrow not\ A_1, \ldots, not\ A_n \in P$$

there exists an A_i such that $P \cup H \vdash A_i$.

If $P \cup H \vdash A_i$ for some A_i, then there is a rule in P of the form

$$A_i \leftarrow not\ B_1, \ldots, not\ B_m$$

such that

$$\{not\ B_1, \ldots, not\ B_m\} \subseteq H$$

Thus, by definition of H and given that P^* is an expansion, there is a clause in $\neg\_P$ of the form

$$A_i \vee \sim not\_B_1 \vee \ldots \vee \sim not\_B_m$$

such that

$$\{not\_B_1, \ldots, not\_B_m\} \subseteq P^*$$

So $P^* \models A_i$ and, because it is an expansion, $\sim not\_A_i \in P^*$.

According to lemma C.0.3, $P^* \models_{CIRC} \sim L$ and, by definition of H, $not\ L \in H$.

Proof of proposition 6.4.1: We begin by proving a lemma:

Lemma C.0.4 *For every noncontradictory default theory Δ, and any context E of Δ :*

$$\Gamma'_{\Delta^s}(E) \subseteq \Gamma'_{\Delta}(E)$$

Proof: In Δ^s every default rule has more literals in the justifications than the corresponding rule in Δ. Thus for every context E, in Δ^s less rules are applicable, and so $\Gamma'_{\Delta^s}(E) \subseteq \Gamma'_{\Delta}(E)$. $\diamondsuit$

Now we prove separately each of the points in the proposition:

1. Let S be the least fixpoint of $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$.

   By lemma C.0.4:

   $$\Gamma'_{\Delta}(S) \supseteq \Gamma'_{\Delta^s}(S)$$

   By the antimonotonicity of Γ'_{Δ^s} (lemma 6.3.3):

   $$\Gamma'_{\Delta^s}(\Gamma'_{\Delta}(S)) \subseteq \Gamma'_{\Delta^s}(\Gamma'_{\Delta^s}(S))$$

   i.e., given that S is by its definition a fixpoint of $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$:

   $$S \subseteq \Gamma'^2_{\Delta^s}(S)$$

   So:

   $$lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}) \subseteq \Gamma'^2_{\Delta^s}(lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}))$$

   i.e. the least fixpoint of $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$ is a pre–fixpoint of $\Gamma'^2_{\Delta^s}$, and thus by the properties of monotonic operators:

   $$lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}) \subseteq lfp(\Gamma'^2_{\Delta^s})$$

2. Again let S be the least fixpoint of $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$, and let $GS = \Gamma'_{\Delta}(S)$.

   By lemma C.0.4:

   $$\Gamma'_{\Delta^s}(GS) \subseteq \Gamma'_{\Delta}(GS)$$

   i.e., by the definition of GS :

   $$\Gamma'_{\Delta^s}(\Gamma'_{\Delta}(S)) \subseteq \Gamma'_{\Delta}(\Gamma'_{\Delta}(S))$$

   So:

   $$lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}) \subseteq \Gamma'^2_{\Delta}(lfp(\Gamma'_{\Delta^s}\Gamma'_{\Delta}))$$

   i.e. the least fixpoint of $\Gamma'_{\Delta^s}\Gamma'_{\Delta}$ is a pre–fixpoint of Γ'^2_{Δ}.

3. Now let S be the least fixpoint of $\Gamma'^2_{\Delta^s}$, and let $GS = \Gamma'_{\Delta^s}(S)$.

   By lemma C.0.4:

   $$\Gamma'_{\Delta^s}(GS) \subseteq \Gamma'_{\Delta}(GS)$$

   i.e., by the definition of GS :

   $$\Gamma'_{\Delta^s}(\Gamma'_{\Delta^s}(S)) \subseteq \Gamma'_{\Delta}(\Gamma'_{\Delta^s}(S))$$

   So:

   $$lfp(\Gamma'^2_{\Delta^s}) \subseteq \Omega(lfp(\Gamma'^2_{\Delta^s}))$$

   i.e. the least fixpoint of $\Gamma'^2_{\Delta^s}$ is a pre–fixpoint of Ω.

4. Finally, let S be the least fixpoint of Γ'^2_Δ.

   By lemma C.0.4:

   $$\Gamma'_\Delta(S) \supseteq \Gamma'_{\Delta^s}(S)$$

   By the antimonotonicity of Γ'_Δ (lemma 6.3.3):

   $$\Gamma'_\Delta(\Gamma'_\Delta(S)) \subseteq \Gamma'_\Delta(\Gamma'_{\Delta^s}(S))$$

   i.e., given that S is by its definition a fixpoint of Γ'^2_Δ :

   $$S \subseteq \Omega(S)$$

   So:

   $$lfp(\Gamma'^2_\Delta) \subseteq \Omega(lfp(\Gamma'^2_\Delta))$$

   i.e. the least fixpoint of Γ'^2_Δ is a pre–fixpoint of Ω.

Proof of theorem 6.6.1: We begin by stating some propositions useful in the sequel.

Proposition C.0.4 *Let $\Delta = (D, \{\})$ be a default theory and E a context such that $\Gamma'_\Delta(E)$ is noncontradictory. Then:*

$$L \in \Gamma'_\Delta(E) \Leftrightarrow \exists \frac{\{b_1,...,b_n\} : \{c_1,...,c_m\}}{L} \in D,\ \forall i,j\ b_i \in \Gamma'_\Delta(E) \wedge \neg c_j \notin E$$

Proof: It is easy to see that under these conditions $\Gamma'_\Delta(E) = \Gamma_\Delta(E)$. Thus the proof follows from properties of the Γ_Δ operator. $\diamondsuit$

Proposition C.0.5 *Let E be an extension of a default theory $\Delta = (D, \{\})$. Then:*

$$L \in \Omega(E) \Leftrightarrow \exists \frac{\{b_1,...,b_n\} : \{\neg c_1,...,\neg c_m\}}{L} \in D \text{ such that}$$

$$\forall i,j,\ b_i \in E \wedge b_i \in \Gamma'_{\Delta^s}(E) \wedge c_j \notin \Gamma'_{\Delta^s}(E).$$

Proof: By definition of Γ'_Δ, and given that $W = \{\}$, it follows from proposition C.0.4 that for $L \in \Omega(E)$ there must exist at least one default in D applied in the second step, i.e. with all prerequesites in $\Omega(E)$ and all negations of justifications not in $\Gamma'_{\Delta^s}(E)$. By hypothesis E is an extension; thus $E = \Omega(E)$ and $E \subseteq \Gamma'_{\Delta^s}(E)$; so for such a rule all prerequesites are in E and in $\Gamma'_{\Delta^s}(E)$, and all negations of justifications are not in $\Gamma'_{\Delta^s}(E)$. $\diamondsuit$

Proposition C.0.6 *Let E be an extension of a default theory $\Delta = (D, \{\})$. Then:*

$$L \notin E \Rightarrow \forall \frac{\{b_1,...,b_n\} : \{\neg c_1,...,\neg c_m\}}{L} \in D,\ \exists i,j,\ b_i \notin E \vee c_j \in \Gamma'_{\Delta^s}(E)$$

Proof: If $L \notin E$ then, given that E is an extension, $L \notin \Omega_\Delta(E)$. Thus no default rule for L is applicable in the second step, i.e. given that $W = \{\}$, and by proposition C.0.4, no rule with conclusion L is such that all its prerequisites are in $\Omega_\Delta(E)$ and no negation of a justification is in $\Gamma'_{\Delta^s}(E)$. $\diamondsuit$

Proposition C.0.7 *Let E be an extension of a default theory $\Delta = (D, \{\})$. Then:*

$$L \notin \Gamma'_{\Delta^s}(E) \Leftrightarrow \forall \frac{\{b_1,...,b_n\} : \{\neg c_1,...,\neg c_m\}}{L} \in D,$$

$$\exists i,j,\ b_i \notin \Gamma'_{\Delta^s}(E) \vee c_j \in E \vee \neg L \in E$$

Proof: Similar to the proof of C.0.6 but now applied to the first step, which imposes the use of seminormal defaults. Thus the need for $\neg L \in E$. $\diamondsuit$

We now prove the main theorem:

($\Rightarrow$) E is a Ω-extension of $\Delta \Rightarrow I$ is a PSM of P.

Here we must prove that for any (objective and default) literal F, $F \in I \Leftrightarrow F \in \Phi(I)$. We do this in three parts: for any objective literal L :

1. $L \in I \Rightarrow L \in \Phi(I)$;
2. $L \notin I \Rightarrow L \notin \Phi(I)$;
3. $not\ L \in I \Leftrightarrow not\ L \in \Phi(I)$.

Each of these proofs proceeds by: translating conditions in I into conditions in E via correspondence; finding conditions in Δ given the conditions in E, and the fact that E is an extension; translating conditions in Δ into conditions in P via correspondence; using those conditions in P to determine the result of operator Φ.

1. Since I corresponds to E and E is a Ω-extension:

$$L \in I \ \Leftrightarrow\ I(L) = 1 \ \Rightarrow\ L \in E \ \Leftrightarrow\ L \in \Omega_\Delta(E)$$

By proposition C.0.5:

$$\begin{array}{rcl} L \in \Omega_\Delta(E) & \Leftrightarrow & \exists \frac{\{b_1,\ldots,b_n\} \,:\, \{\neg c_1,\ldots,\neg c_m\}}{L} \in D, \\ & & \forall i, b_i \in E \wedge b_i \in \Gamma'_{\Delta^s}(E)\ and \\ & & \forall j, c_j \notin \Gamma'_{\Delta^s}(E). \end{array}$$

By translating, via the correspondence definitions, the default and the conditions on E into a rule and conditions on I:

$$\begin{array}{rcl} L \in E & \Rightarrow & \exists L \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots,c_m \in P, \\ & & \forall i, I(b_i) = 1\ and\ \forall j, I(c_j) = 0 \\ & \Rightarrow & L \in least\left(\frac{P}{I}\right) \end{array}$$

by properties of $least\left(\frac{P}{I}\right)$.
Given that the operator Coh does not delete literals from I :

$$L \in I \Rightarrow L \in \Phi(I).$$

2. Since I corresponds to E :

$$L \notin I \Leftrightarrow L \notin E.$$

By proposition C.0.6:

$$L \notin E \Rightarrow \forall \frac{\{b_1,\ldots,b_n\} \ :\ \{\neg c_1,\ldots,\neg c_m\}}{L} \in D$$

where either a $b_i \notin E$ or a $c_j \in \Gamma'_{\Delta^s}(E)$, .
Translating, via the correspondence definitions, the default and the conditions on E into a rule and conditions on I :

$$\begin{array}{rcl} L \notin E & \Rightarrow & \forall L \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots,c_m \in P, \\ & & \exists i, j\ I(b_i) \neq 1 \vee I(c_j) \neq 0 \\ & \Rightarrow & L \notin least\left(\frac{P}{I}\right) \end{array}$$

by properties of $least\left(\frac{P}{I}\right)$.

Given that the operator Coh does not add objective literals to I :

$$L \notin I \Rightarrow L \notin \Phi(I).$$

3. Given that E corresponds to I :

$$not\ L \in I \Leftrightarrow L \notin \Gamma'_{\Delta^s}(E)$$

By proposition C.0.7:

$$\begin{array}{rcl} L \notin \Gamma'_{\Delta^s}(E) & \Leftrightarrow & \forall \frac{\{b_1,\ldots,b_n\} \,:\, \{\neg c_1,\ldots,\neg c_m\}}{L} \in D, \\ & & \exists i,j\ b_i \notin \Gamma'_{\Delta^s}(E) \vee c_j \in E \vee \neg L \in E \end{array}$$

Translating into logic programs:

$$\begin{array}{rcl} L \notin \Gamma'_{\Delta^s}(E) & \Leftrightarrow & \forall L \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots,c_m \in P, \\ & & (\exists i,j\ I(b_i) = 0 \vee I(c_j) = 1) \vee \neg L \in E. \end{array}$$

By properties of the least operator.

$$not\ L \in I \Leftrightarrow not\ L \in least\left(\frac{P}{I}\right) \vee \neg L \in E \quad (*)$$

It was proven before that:

$$\begin{array}{rcl} \neg L \in E & \Leftrightarrow & \exists \neg L \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots,not\ c_m \in P, \\ & & \exists i,j\ I(b_i) = 1 \vee I(c_j) = 0. \end{array}$$

By properties of $least\frac{P}{I}$:

$$\neg L \in E \Leftrightarrow \neg L \in least\left(\frac{P}{I}\right)$$

Using correspondence, we can simplify the equivalence $(*)$ to:

$$\begin{array}{rcl} not\ L \in I & \Leftrightarrow & not\ L \in least\left(\frac{P}{I}\right) \vee \neg L \in least\left(\frac{P}{I}\right) \Leftrightarrow \\ & \Leftrightarrow & not\ L \in \Phi(I) \end{array}$$

this last equivalence being due to the definitions of operators Coh and Φ.

($\Leftarrow$) I is a PSM of $P \Rightarrow E$ is a Ω-extension of T.

By definition of correspondence between interpretations and contexts, it is easy to see that E is consistent and $E \subseteq \Gamma'_{\Delta^s}(E)$. So we only have to prove that $E = \Omega_\Delta(E)$. We do this by proving that:

$$\forall L\ L \in E \Leftrightarrow L \in \Omega_\Delta(E).$$

By definition of corresponding context:

$$L \in E \Leftrightarrow I(L) = 1$$

Since I is a PSM of P :

$$\begin{array}{rcl} I(L) = 1 & \Leftrightarrow & \exists L \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots,not\ c_m \in P, \\ & & \forall i\ I(b_i) = 1\ and\ \forall j\ I(c_j) = 0 \end{array}$$

where $n, m \geq 0$.

By translating, via the correspondence definitions, the rule and the conditions on I into a default and conditions on E:

$$\begin{aligned} I(L) = 1 \quad \Leftrightarrow \quad & \exists \frac{\{b_1,\ldots,b_n\} : \{\neg c_1,\ldots,\neg c_m\}}{L} \in D, \\ & \forall i \; b_i \in E \wedge b_i \in \Gamma'_{\Delta^s}(E) \; and \\ & \forall j \; c_j \notin E \wedge c_j \notin \Gamma'_{\Delta^s}(E) \end{aligned}$$

Given that such a rule exists under such conditions, it follows easily from proposition C.0.4 that:

$$L \in E \Leftrightarrow L \in \Omega_\Delta(E)$$

Proof of theorem 7.4.1:

1. Let $\mathcal{C}$ be a (possibly infinite) set of complete scenaria, i.e.

$$\mathcal{C} \subseteq CS_P \quad \text{and} \quad \mathcal{C} \neq \{\}.$$

Let $\mathcal{C}_\downarrow$ be the set of all admissible scenaria contained in all scenaria of $\mathcal{C}$, and let S_0 be the union of all elements in $\mathcal{C}_\downarrow$.

Since that:

$$\forall S \in \mathcal{C} \mid S_0 \subseteq S$$

it is clear that S_0 is admissible. It remains to prove that S_0 is also complete.

Let *not* L be a literal acceptable wrt S_0. Then $S' = S_0 \cup \{not\ L\}$ is again admissible and so, by definition, $S' \in \mathcal{C}_\downarrow$. Thus *not* $L \in S_0$.

If *not* L is mandatory wrt S_0 then, since S_0 is admissible, *not* $L \in S_0$. Thus S_0 is complete.

2. The proof of this point is obvious given the previous one.

3. The program in example 7.14 shows that in general a maximal element might not exist.

Proof of lemma 7.4.7: Without loss of generality (by theorem 10.1.2) we consider that P is in semantic kernel form, i.e. P is a set of rules of the form:

$$L \leftarrow not\ A_1, \ldots, not\ A_n \qquad n \geq 0$$

Let $S = T \cup not\ F$ be a PSM of P, i.e. (according to the equivalent definition of $PSMs$ (theorem 6.7.1) in section 6.7):

(i) $T = \Gamma\Gamma_s T$

(ii) $T \subseteq \Gamma_s T$

(iii) $\nexists L \mid \{L, \neg L\} \subseteq T$

and additionally $F = \{L \mid L \notin \Gamma_s T\}$.

Let $H = \{not\ L \mid L \notin \Gamma_s T\}$[3]. We prove that $P \cup H$ is a complete scenario, i.e. for all *not* L :

1. $not\ L \in H \Rightarrow P \cup H \nvdash L$
2. $not\ L \in H \Rightarrow not\ L \in Mand(H) \vee not\ L \in Acc(H)$
3. $not\ L \in Mand(H) \Rightarrow not\ L \in H$
4. $not\ L \in Acc(H) \Rightarrow not\ L \in H$

This proof is accomplished by proving separately each of the conditions above.

1. By definition of H :

$$not\ L \in H \Leftrightarrow L \notin \Gamma_s T.$$

Given that the P is in the semantic kernel form:

$$L \notin \Gamma_s T \Rightarrow \neg L \in T \vee \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i \in T$$

[3] I.e. $H \equiv not\ F$.

Let us assume the first disjunct:

$$\neg L \in T \overset{\text{by (iii)}}{\Rightarrow} L \notin T \overset{\text{by (i)}}{\Rightarrow} L \notin \Gamma\Gamma_s T$$

Again because P is in semantic kernel form:

$$\text{1st disjunct} \Rightarrow \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i \in \Gamma_s T$$

By definition of H :

$$\text{1st disjunct} \Rightarrow \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists not\ A_i \notin H \overset{\text{trivial}}{\Rightarrow} P \cup H \nvdash L$$

Now let us assume the second disjunct; then:

$$A_i \in T \overset{\text{by (ii)}}{\Rightarrow} A_i \in \Gamma_s T \overset{\text{by def of H}}{\Rightarrow} not\ A_i \notin H$$

Thus:

$$\text{2nd disjunct} \Rightarrow \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists not\ A_i \notin H \overset{\text{trivial}}{\Rightarrow} P \cup H \nvdash L$$

2. As proven at the begining of 1 above:

$$not\ L \in H \Rightarrow \neg L \in T \vee \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i \in T$$

Let us assume the first disjunct:

$$\neg L \in T \overset{\text{by (i)}}{\Rightarrow} \neg L \in \Gamma\Gamma_s T \Rightarrow \exists \neg L \leftarrow not\ B_1, \ldots, not\ B_m \mid \forall B_i,\ B_i \notin \Gamma_s T$$

By definition of H, $B_i \notin \Gamma_s T \Rightarrow not\ B_i \in H$. Thus trivially:

$$\text{1st disjunct} \Rightarrow P \cup H \vdash \neg L \overset{\text{by def of Mand(H)}}{\Rightarrow} not\ L \in Mand(H)$$

Now let us assume the second disjunct:

$$A_i \in T \overset{\text{by (i)}}{\Rightarrow} A_i \in \Gamma\Gamma_s T$$

$$\text{2nd disjunct} \Rightarrow \exists A_i \leftarrow not\ C_1, \ldots, not\ C_k \mid \forall C_j,\ C_j \notin \Gamma_s T \Rightarrow P \cup H \vdash A_i$$

Thus the second disjunct implies:

$$\forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i,\ P \cup H \vdash A_i \overset{\text{by def of Acc(H)}}{\Rightarrow} not\ L \in Acc(H)$$

3. By definition of $Mand(H)$:

$$not\ L \in Mand(H) \Rightarrow \exists \neg L \leftarrow not\ A_1, \ldots, not\ A_n \mid \forall not\ A_i,\ not\ A_i \in H$$

By definition of H, $not\ A_i \in H \Leftrightarrow A_i \notin \Gamma_s T$. Thus:

$$\text{hyp.} \Rightarrow \neg L \in \Gamma\Gamma_s T \overset{\text{by (i)}}{\Rightarrow} \neg L \in T \overset{\text{by seminormallity}}{\Rightarrow} L \notin \Gamma_s T \Leftrightarrow not\ L \in H$$

4. By definition of $Acc(H)$:

$$not\ L \in Acc(H) \Rightarrow \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i,\ P \cup H \vdash A_i.$$

Now:

$$P \cup H \vdash A_i \Rightarrow \exists A_i \leftarrow not\ B_1, \ldots, not\ B_m \mid \forall not\ B_j,\ not\ B_j \in H$$

By definition of H :

$$not\ B_j \in H \Rightarrow B_j \in \Gamma_s T$$

Thus $A_i \in \Gamma\Gamma_s T$, and by (i) $A_i \in T$. So:

$$not\ L \in Acc(H) \Rightarrow \forall l \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists not\ A_i,\ A_i \in T \Rightarrow$$

$$\Rightarrow L \notin \Gamma_s T \Rightarrow not\ L \in H$$

Proof of lemma 7.4.8: Given that $P \cup H$ is a complete scenario then, for all $not\ L$:

(i) $not\ L \in H \Rightarrow P \cup H \nvdash \neg L$

(ii) $not\ L \in H \Rightarrow not\ L \in Mand(H) \vee not\ L \in Acc(H)$

(iii) $not\ L \in Mand(H) \Rightarrow not\ L \in H$

(iv) $not\ L \in Acc(H) \Rightarrow not\ L \in H$

Let $S = \{L \mid P \cup H \vdash L\}$. According to theorem 6.7.1 we must prove that:

1. $\forall L \mid L \in S \Rightarrow \neg L \notin S$
2. $S \subseteq \Gamma_s S$
3. $S = \Gamma\Gamma_s S$
4. $H = \{not\ L \mid L \notin \Gamma_s S\}$

In order to prove this lemma we begin by proving that

$$\Gamma_s S = \{L \mid not\ L \notin H\} \tag{C.1}$$

This is achieved by proving (where $U = \{L \mid not\ L \notin H\}$):

(a) $\forall L \mid L \notin \Gamma_s S \Rightarrow L \notin U$

(b) $\forall L \mid L \notin U \Rightarrow L \notin \Gamma_s S$

(a) By definition of Γ_s :

$$L \notin \Gamma_s S \Rightarrow \neg L \in S \vee \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i \in S$$

We prove that both disjuncts imply $L \notin U$:

$$\neg L \in S \overset{\text{def of S}}{\Rightarrow} P \cup H \vdash \neg L \overset{\text{def of Mand(H)}}{\Rightarrow} not\ L \in Mand(H)$$

Since $P \cup H$ is a complete scenario:

$$not\ L \in Mand(H) \Rightarrow not\ L \in H \overset{\text{def of U}}{\Rightarrow} \text{Ł} \notin U$$

Since:

$$A_i \in S \overset{\text{def of S}}{\Rightarrow} P \cup H \vdash A_i$$

the second disjunct implies:

$$\forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i,\ P \cup H \vdash A_i \overset{\text{def of Acc(H)}}{\Rightarrow} not\ L \in Acc(H)$$

Given that $P \cup H$ is a complete scenario:

$$not\ L \in Acc(H) \Rightarrow not\ L \in H \Leftrightarrow \text{L} \notin H$$

(b) By definition of U :

$$L \notin U \Leftrightarrow not\ L \in H$$

Since H is complete scenario then by (ii):

$$L \notin U \Rightarrow P \cup H \vdash \neg L \vee \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i,\ P \cup H \vdash A_i$$

Both disjuncts lead to the conclusion that $L \notin \Gamma_s S$:

- $P \cup H \vdash \neg L \overset{\text{def of S}}{\Rightarrow} \neg L \in S \overset{\text{by seminormallity}}{\Rightarrow} L \notin \Gamma_s S$
- $P \cup H \vdash A_i \overset{\text{def of S}}{\Rightarrow} A_i \in S \Rightarrow L \notin \Gamma_s S$

Now we prove the four points above:

1. Trivial because $P \cup H$ is a consistent scenario.
2. Since $\Gamma_s S = U$, we have to prove that:

   $$S = \{L \mid P \cup H \vdash L\} \subseteq \{L \mid not\ L \notin H\} = \Gamma_s S$$

   which is also trivial because $P \cup H$ is a consistent scenario.
3. The proof of this point is divided into two parts.
   (a) $\forall L \mid L \notin \Gamma\Gamma_s S \Rightarrow L \notin S$, i.e. $S \subseteq \Gamma\Gamma_s S$.

   $$L \notin \Gamma\Gamma_s S \Rightarrow \forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists A_i \in \Gamma_s S$$

   If $A_i \in \Gamma_s S$ then by equivalence (C.1) above $not\ A_i \notin H$. Thus:

   $$\forall L \leftarrow not\ A_1, \ldots, not\ A_n \mid \exists not\ A_i \notin H \Rightarrow P \cup H \nvdash L \overset{\text{def of S}}{\Rightarrow} L \notin S$$

   (b) $\forall L \mid L \in \Gamma\Gamma_s S \Rightarrow L \in S$, i.e. $S \supseteq \Gamma\Gamma_s S$. Thus:

   $$L \in \Gamma\Gamma_s S \Rightarrow \exists L \leftarrow not\ A_1, \ldots, not\ A_n \mid \forall A_i\ A_i \notin \Gamma_s S$$

   If $A_i \notin \Gamma_s S$ then by equivalence (C.1) above $not\ A_i \in H$. Thus:

   $$\exists L \leftarrow not\ A_1, \ldots, not\ A_n \mid \forall not\ A_i,\ not\ A_i \in H$$

   $$\Rightarrow P \cup H \vdash L \overset{\text{def of S}}{\Rightarrow} L \in S$$
4. From equation C.1, for every objective literal L :

   $$L \in \Gamma_s S \Leftrightarrow not\ L \notin H$$

   or equivalently:

   $$L \notin \Gamma_s S \Leftrightarrow not\ L \in H$$

   i.e.

   $$H = \{not\ L \mid L \notin \Gamma_s S\}$$

Proof of lemma 9.1.1: We prove that for all $not\ a' \in B(P)$, if $\langle A; WFM(P+A)\rangle$ is inconsistent then

$$\langle A \cup \{not\ a'\}; WFM(P + (A \cup \{not\ a'\}))\rangle$$

is also inconsistent.

By definition of inconsistent A–model:

$$\exists not\ b \in A \mid b \in WFM(P+A)$$

so it suffices to guarantee that:

$$b \notin WFM(P + (A \cup \{not\ a'\})) \Rightarrow a' \in WFM(P + (A \cup \{not\ a'\})).$$

Consider $b \notin WFM(P+A\cup\{not\ a'\})$. Since b is true in $P+A$, and since $P+(A\cup\{not\ a'\})$ only differs from $P+A$ in rules with a' or $not\ a'$ in the body, it follows that there is a support set $SS_{P+A}(b)$ containing a' (in appendix B we recall the definition of support set introduced in [Pereira *et al.*, 1991a]), and thus, by definition of support set, a' is also true in $P+A$.

Since $a' \in WFM(P+A)$, by propositions B.0.1 and B.0.2 there is a support set $SS_{P+A}(a')$ (according to the definition B.0.1 of support set for normal programs) such that:

$$a' \notin SS_{P+A}(a') \ \text{ and } \ not\ a' \notin SS_{P+A}(a').$$

As the addition of $not\ a'$ to $P+A$ only changes rules with $not\ a'$ or a', then:

$$Rules(SS_{P+A}(a')) \subseteq P + (A \cup \{not\ a'\}) \subseteq P + A$$

and by proposition B.0.3 $a' \in WFM(P + (A \cup \{not\ a'\}))$.

Proof of lemma 9.2.1: First let us assume that L is an objective literal.

If $L \in WFSX(P)$ then there exists a rule in P

$$L \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

such that

$$\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\} \subseteq WFSX(P)$$

If this rule is not deleted in $P+A$, i.e.

$$\{not\ B_1, \ldots, not\ B_n\} \cap A = \{\}$$

then this theorem applies recursively[4] and so:

$$\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\} \subseteq WFSX(P+A)$$

Given that the rule is not deleted:

$$L \leftarrow B_1, \ldots, B_n, not\ D_1, \ldots, not\ D_k \in P + A$$

where

$$\{not\ D_1, \ldots, not\ D_k\} \subseteq \{not\ C_1, \ldots, not\ C_m\}$$

Thus $L \in WFSX(P+A)$.

If the rule is deleted then there exists in A one $not\ B_i$ where B_i is one elements of the rule body. Applying this theorem recursively for B_i it follows that $B_i \in WFSX(P+A)$, i.e.

[4]The base conditions for the recursion are similar to those of proposition 9.1.4 and are omitted for brevity.

$\langle A; WFSX(P + A)\rangle$ is not consistent, contradicting this way one of our hypotheses.

If the literal is of the form *not* L the proof follows along the same lines of the proof of proposition 9.1.4, and is omitted.

Proof of theorem 9.2.3: We have to prove that for any two consistent A–models $\langle A; M\rangle$ and $\langle A'; M'\rangle$, their meet $\langle A \cap A'; M''\rangle$ exists, is unique, and is a consistent A–model. The uniqueness is guaranteed by definition of A–model, given the uniqueness of *WFSX* of any program.

We begin by proving (by contradiction) that $P+(A\cap A')$ is noncontradictory, i.e. $\langle A\cap A'; M''\rangle$ is an A–model.

Assume that $P + (A \cap A')$ is contradictory. Then there are in $P + (A \cap A')$ at least two rules with complementary heads whose bodies are true in the paraconsistent *WFSX*. If none of these rules is deleted in one of $P + A$ or $P + A'$ then either $P + A$ or $P + A'$ is contradictory. If one of the rules is deleted, say in $P + A$, then there exists in the body of that rule one B_i such that *not* $B_i \in A$. Since B_i is true in $P + (A \cap A')$, similarly to the proof of lemma 9.2.1 one can prove that $B_i \in WFSX(P + A)$, i.e. $\langle A; M\rangle$ is inconsistent, thus contradicting one of our hypotheses.

Following the lines of the proof of lemma 9.1.1, but using the definitions of support for extended programs (definition 8.3.14) instead of that for normal programs, it is easy to prove that if an A–model is inconsistent then all A–models greater than it are also inconsistent. Given this, and on the assumption that $\langle A; M\rangle$ and $\langle A'; M'\rangle$ are consistent, it follows that $\langle A \cap A'; M''\rangle$ is also consistent.

Proof of lemma 9.2.4: Let:

$$A_J = \bigcup_{k\in K} A_k$$

We have to prove that:

1. $P + A_J$ is noncontradictory.

   - By contradiction, assume that $P + A_J$ is contradictory. Then by definition all maximal A–models are untenable (contradiction).

2. There exists no *not* $L \in A_J$ such that $L \in WFSX(P + A_J)$.

   - Again by contradiction, assume that there exists one *not* $L \in A_J$ such that $L \in WFSX(P + A_J)$. If *not* $L \in A_J$ then there exist one A–model $\langle A_i; M_i\rangle$, maximal in CS, such that *not* $L \in A_i$. Since $L \in WFSX(P + A_J)$, $\langle A_i; M_i\rangle$ is untenable (contradiction).

3. There exists no consistent A–model $\langle A; M\rangle$ such that $L \in M$ and *not* $L \in A_J$.

   - By definition of candidate structure, every A–models $\langle A_i; M_i\rangle$ in CS is sustainable. Thus there is no A–model $\langle A; M\rangle$ such that $L \in M$ and *not* $L \in A_i$ and so, given that this holds for all A_i in CS, there is no A–model $\langle A; M\rangle$ such that $L \in M$ and

     $$not\ L \in \bigcup_{k\in K} A_i$$

     i.e. the join is sustainable.

Proof of theorem 10.1.3:

Let P be an extended logic program and let P' be the program obtain from P by replacing the rule r of P

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

by the rule r' :

$$H \leftarrow not\ \neg B_1, B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

We begin by proving that if M is a PSM of P, and B_1 and $\neg B_1$ are not both undefined in M[5] then $\frac{P}{M} = \frac{P'}{M}$[6]. Since the modulo transformation is made rule by rule, and $P - r = P' - r'$, to prove that it is enough to show that $\frac{r}{M} = \frac{r'}{M}$.

Assume that M is a PSM of P. Then:

- If $B_1 \in M$ then, since M is a PSM and thus also an interpretation, $not\ \neg B_1 \in M$, and so the default literal $not\ \neg B_1$ is deleted from r' in $\frac{r'}{M}$. Thus, trivially, $\frac{r'}{M} = \frac{r}{M}$.
- If $\neg B_1 \in M$, then it is clear that the rule is deleted in both cases, and so $\frac{r}{M} = \frac{r'}{M} = \{\}$.
- If $not\ B_1 \in M$ then both rules again are deleted.
- If $not\ \neg B_1 \in M$ then the literal is deleted from r', and so $\frac{r'}{M} = \frac{r}{M}$.
- Since we are assuming that B_1 and $\neg B_1$ are not both undefined in M, no other case can occur.

So for these cases:

$$M = Coh\left(least\left(\frac{P}{M}\right)\right) \quad \Rightarrow \quad M = Coh\left(least\left(\frac{P'}{M}\right)\right)$$

i.e. M is a PSM of P'.

Now let M be a PSM of P', such that:

$$M \cap \{B_1, \neg B_1, not\ B_1, not\ \neg B_1\} = \{\}$$

If the rule r is deleted in $\frac{P}{M}$, then it is clear that r' is also deleted in $\frac{P'}{M}$, and so $\frac{P}{M} = \frac{P'}{M}$. Thus we are in a similar situation as the one before, and M is a PSM of P.

Otherwise let:

$$\frac{r}{M} = H \leftarrow B_1, \ldots, B_n, not\ C_i, \ldots, not\ C_j$$

where $\{not\ C_i, \ldots, not\ C_j\} \subseteq \{not\ C_1, \ldots, not\ C_m\}$. Then:

$$\frac{r'}{M} = H \leftarrow \mathbf{u}, B_1, \ldots, B_n, not\ C_i, \ldots, not\ C_j$$

From the definition of the *least* operator, and the hypothesis that B_1 is undefined in M, it is clear that the rule $\frac{r}{M}$ does not provide a way of proving H in $\frac{P}{M}$ and $not\ H$ does not belong to $least\left(\frac{P}{M}\right)$.

Since $\mathbf{u}$ is in the body of $\frac{r'}{M}$ the same happens in $\frac{P'}{M}$.

From this result it follows trivially that also in this case M is a PSM of P'. So in every case, if M is a PSM of P it is also a PSM of P'.

---

[5] I.e. $M \cap \{B_1, \neg B_1, not\ B_1, not\ \neg B_1\} \neq \{\}$.

[6] Where $\frac{P}{M}$ is as in definition 4.2.1.

The proof that if M is a PSM of P' it is also a PSM of P, is quite similar and is omitted for brevity.

The proof that "P is contradictory iff P' is contradictory" follows directly from the proof above. Note that the statement is equivalent to "P is noncontradictory iff P' is noncontradictory".

If P is noncontradictory then it has at least one PSM. Let M be one PSM of P. Then, as proven above, M is also a PSM of P', i.e. P' is noncontradictory. Similarly, if P' is noncontradictory P is also noncontradictory.